

USENIX

Conference Proceedings

Summer
1987

Phoenix,
Arizona

USENIX

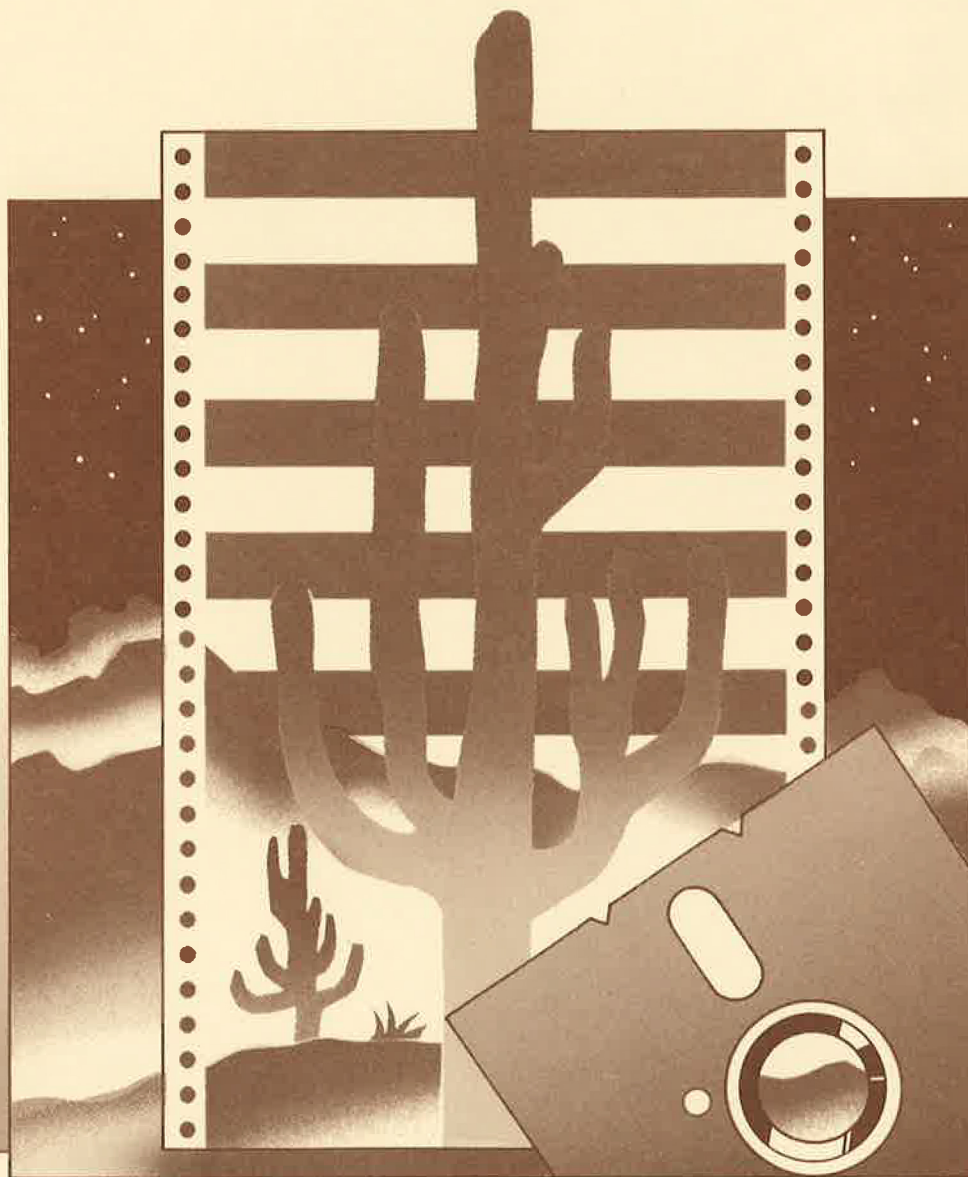
The Professional and
Technical UNIX® Association

UNIX® is a registered
Trademark of AT&T

CONFERENCE PROCEEDINGS

Summer 1987
USENIX
Technical Conference
and Exhibition

June 8-12, 1987 Phoenix, Arizona



USENIX Association

**Proceedings of the
Summer 1987 USENIX Conference**

**June 8-12, 1987
Phoenix, Arizona USA**

For additional copies of these proceedings, write:

USENIX Association

P.O. Box 2299

Berkeley, CA 94701 USA

Price: \$20.00 plus \$25.00 for overseas mail

Copyright 1987 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain
with the author or the author's employer.

UNIX is a trademark of AT&T Bell Laboratories.

Other trademarks are noted in the text.

ACKNOWLEDGEMENTS

Sponsored by:	USENIX Association P.O. Box 2299 Berkeley, CA 94710	
Program Chair:	Eric Allman	Britton Lee, Inc.
Program Committee:	Greg Chesson Thomas Ferrin Daniel Klein Sam Leffler Jay Lepreau John Mashey Evi Nemeth John Quarterman Dennis Ritchie Dave Taylor Chris Torek	Silicon Graphics, Inc. Univ. of CA, San Francisco CMU Software Engineering Institute Pixar Univ. of Utah MIPS Computer Systems, Inc. Univ. of Colorado TIC AT&T Bell Laboratories Hewlett Packard Laboratories Univ. of Maryland
Proceedings Production:	Eric Allman Paul Kooros Evi Nemeth	Britton Lee, Inc. Univ. of Colorado Univ. of Colorado
Tutorial Coordinator:	John Donnelly	USENIX Association
USENIX Meeting Planner:	Judith F. DesHarnais	USENIX Association
Vendor Exhibition Manager:	John Donnelly	USENIX Association

TABLE OF CONTENTS

PLENARY SESSION

Wednesday (8:30-10:30) (Ballroom) Chair: Eric Allman

KEYNOTE ADDRESS: Why We Have To Make UNIX Invisible
Steven Jobs, NeXT, Inc.

The Diamond Multimedia Editor	1
<i>Terrence Crowley, Harry Forsdick, Matt Landau, Virginia Travers, BBN Laboratories, Inc.</i>	

DOCUMENT PREPARATION

Wednesday (11:00-12:30) (Ballroom) Chair: Greg Chesson

An Interactive WYSIWYG Table Editor	19
<i>S.L. Murrel, AT&T Bell Laboratories; D. DeBaer, University of Antwerp</i>	
PSFIG – A DITROFF Preprocessor for Postscript Figures	31
<i>Ned Batchelder & Trevor Darrell, University of Pennsylvania</i>	
An Environment for SGML Document Preparation	43
<i>Le van Huu, University of Milan</i>	

MEMORY MANAGEMENT

Wednesday (2:00-3:30) (Ballroom) Chair: Dennis Ritchie

A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach	53
<i>Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young, David B. Golub, Mary R. Thompson, William Bolosky, & Richard Sanzi, Carnegie-Mellon University</i>	
A Replacement for Berkeley Memory Management	69
<i>Pervaze Akhtar, Gould Computer Systems Division</i>	
Virtual Memory Architecture in SunOS	81
<i>Robert A. Gingell, Joseph P. Moran, & William A. Shannon, Sun Microsystems, Inc.</i>	

REAL WORLD

Wednesday (2:00-3:30) (Tucson Rooms) Chair: Dave Taylor

CAS Perspective on the Maturation of UNIX	95
<i>Susan A. Funk, Chemical Abstracts Service</i>	
Keeping Watch Over the Flocks by Night (and Day)	105
<i>Kenneth Ingham, University of New Mexico</i>	
X.400 Messaging on UNIX	111
<i>Andrew Draskoy & Gerald Neufeld, University of British Columbia</i>	

PROGRAMMING SYSTEMS

Wednesday (4:00-5:30) (Ballroom) Chair: Chris Torek

The X Toolkit – The Standard Toolkit for X Version 11	117
<i>Ram Rao & Smokey Wallace, DEC Ultrix Engineering Group</i>	
Shared Libraries in SunOS	131
<i>Robert A. Gingell, Meng Lee, Xuong T. Dang, & Mary S. Weeks, Sun Microsystems, Inc.</i>	
A Debugger-Based System for Graphical Display and Editing of Data Structures	147
<i>Peter Potrebic & Phil Goldman, Apple Computer</i>	

WORK IN PROGRESS

Wednesday (4:00-5:30) (Tucson Rooms) Chair: Debbie Scherrer

Ten minute presentations of current work.

PROCESS MODELS

Thursday (9:00-10:30) (Ballroom) Chair: Tom Ferrin

A New IPC System for Bitmap Graphics Applications: Review, Model, and Benchmarks	159
<i>C.D. Blewett, M. Wish, & J.I. Helfman, AT&T Bell Laboratories</i>	
Mach Threads and the UNIX Kernel: The Battle for Control	185
<i>Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, & Michael W. Young, Carnegie-Mellon University</i>	
A Dynamically Extensible Streams Implementation	199
<i>Jim Rees, Margaret Olson, & J. Sasidhar, Apollo Computer, Inc.</i>	

ARCHITECTURE

Thursday (11:00-12:30) (Ballroom) Chair: John Mashey

Protocol Engine Design	209
<i>Greg Chesson, Silicon Graphics</i>	
Virtual Address Cache in UNIX	217
<i>Ray Cheng, Sun Microsystems, Inc.</i>	
UNIX on a VLIW	225
<i>Patrick Clancy, Benjamin F. Cutler, J. Christopher Dodd, Douglas W. Gilmore, Robert P. Nix, John J. O'Donnell, & Christopher P. Ryland, Multiflow Computer, Inc.</i>	

SECURITY

Thursday (11:00-12:00) (Tucson Rooms) Chair: Evi Nemeth

UNIX Without the Superuser	243
<i>M.S. Hecht, M.E. Carson, C.S. Chandrasekaran, R.S. Chapman, L.J. Dotterer, V.D. Gligor, W.D. Jiang, A. Johri, G.L. Luckenbaugh, & N. Vasudevan, IBM Federal Systems Division</i>	
Partial Model for a B-Level UNIX	257
<i>Frank Knowles, Gould Computer Systems Division</i>	

FILE SYSTEMS

Thursday (2:00-3:30) (Ballroom) Chair: Kirk McKusick

A Remote-File Cache for RFS	273
<i>M.J. Bach, M.W. Luppi, & A.S. Melamed, AT&T Bell Laboratories; K. Yueh, AT&T Information Systems</i>	
RFS in SunOS	281
<i>Howard Chartock, Sun Microsystems, Inc.</i>	
GFS Revisited -or- How I Lived with Four Different Local File Systems	291
<i>Matt Koehler, DEC Ultrix Engineering Group</i>	

INTERNATIONALIZATION

Thursday (2:00-3:00) (Tucson Rooms) Chair: Joe Kalash

Now, UNIX Talks To Me In My Language	307
<i>Pascal Beyls, BULL</i>	
A UNIX System V STREAMS TTY Implementation for Multiple Language Processing	323
<i>Hiromichi Kogure & Richard McGowan, AT&T UNIX Pacific Co., Ltd.</i>	

COMPILERS

Thursday (4:00-5:30) (Ballroom) Chair: Donn Seeley

Automatic Error Recovery in a Fast Parser	337
<i>Robert W. Gray, University of Colorado</i>	
Cross-Module Optimizations: Its Implementation and Benefits	347
<i>Mark I. Himelstein, Fred C. Chow, & Kevin Enderby, MIPS Computer Systems</i>	
RPCC - A Stub Compiler for Sun RPC	357
<i>Irving Reid, University of Saskatchewan</i>	

WORK IN PROGRESS

Thursday (4:00-5:30) (Tucson Rooms) Chair: David Yost

Ten minute presentations of current work.

NETWORKS

Friday (9:00-10:30) (Ballroom) Chair: Jay Lepreau

Implementing the Reliable Data Protocol (RDP)	367
<i>Craig Partridge, Harvard University/BBN Laboratories, Inc.</i>	
Remote UNIX, Turning Idle Workstations into Cycle Servers	381
<i>Michael J. Litzkow, University of Wisconsin</i>	
The Networking Computing Architecture and System: An Environment for Developing Distributed Applications	385
<i>Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, & Geoffrey L. Wyant, Apollo Computer, Inc.</i>	

PERFORMANCE

Friday (11:00-12:00) (Ballroom) Chair: John Quarterman

Solving Performance Problems on a Multiprocessor UNIX System	399
<i>T.P. Lee, AT&T Bell Laboratories; M.W. Luppi & R.E. Menninger, AT&T Information Systems</i>	
Taking Performance Evaluation Out of the "Stone" Age	407
<i>Ken J. McDonell, Monash University</i>	

PANEL – UNIX DIRECTIONS

Friday (11:00-12:30) (Tuscon Rooms) Chair: Rob Kolstad

The UNIX Marketplace in 1987: Life, the UNiverse and Everything	419
<i>Andrew Tannenbaum, Interactive Systems Corp.</i>	
Unix at the Turn of the Century	425
<i>Michael Tilson, HCR Corporation</i>	

GRAB BAG

Friday (2:00-4:00)

(Ballroom)

Chair: Daniel Klein

UTek Build Enironment	437
<i>Alan McIvor, Tektronix Graphics Workstation Division</i>	
Mk: A Successor to Make	445
<i>Andrew Hume, AT&T Bell Laboratories</i>	
Miranda – An Advanced Functional Programming System Running Under UNIX	459
<i>David Turner, University of Kent</i>	
Experiences with DREGS	470
<i>Allan Bricker, Morgan Clark, Tad Lebeck, Barton P. Miller, & Peter Wu, University of Wisconsin</i>	

STUDENT PAPER AWARDS

Like UNIX, the Usenix Association has always thrived on student participation. This conference is no exception. In recognition of this, USENIX has instituted awards for the best student papers.

At this conference, the two award winners are Robert Gray for "Automatic Error Recovery in a Fast Parser" and Irving Reid for "RPCC — A Stub Compiler for Sun RPC." Both papers are being presented Thursday afternoon in the "Compilers" session.

It is the intention of the USENIX Board to make similar awards at the Winter 1988 meeting in Dallas. Submissions to that conference should indicate student status.

Peter H. Salus
Executive Director
USENIX Association

AUTHOR INDEX

Pervaze Akhtar	69	V.D. Gligor	243	S.L. Murrel	19
M.J. Bach	273	Phil Goldman	147	Gerald Neufeld	111
Ned Batchelder	31	David B. Golub	53	Robert P. Nix	225
Pascal Beyls	307	David B. Golub	185	John J. O'Donnell	225
David L. Black	185	Robert W. Gray	337	Margaret Olson	199
C.D. Blewett	159	M.S. Hecht	243	Craig Partridge	367
William Bolosky	53	J.I. Helfman	159	Joseph N. Pato	385
Allan Bricker	470	Andrew Hume	445	Peter Potrebic	147
M.E. Carson	243	Mark I. Himelstein	347	Ram Rao	117
C.S. Chandrasekaran	243	Le van Huu	43	Richard F. Rashid	53
R.S. Chapman	243	Kenneth Ingham	105	Richard F. Rashid	185
Howard Chartock	281	W.D. Jiang	243	Jim Rees	199
Ray Cheng	217	A. Johri	243	Irving Reid	357
Greg Chesson	209	Frank Knowles	257	Christopher P. Ryland	225
Fred C. Chow	347	Matt Koehler	291	Richard Sanzi	53
Patrick Clancy	225	Hiromichi Kogure	323	J. Sasidhar	199
Morgan Clark	470	Matt Landau	1	William A. Shannon	81
Eric Cooper	185	Paul J. Leach	385	Andrew Tannenbaum	419
Terrence Crowley	1	Tad Lebeck	470	Avadis Tevanian Jr.	53
Benjamin F. Cutler	225	Meng Lee	131	Avadis Tevanian Jr.	185
Xuong T. Dang	131	T.P. Lee	399	Mary R. Thompson	53
Trevor Darrell	31	Michael J. Litzkow	381	Michael Tilson	425
D. DeBaer	19	G.L. Luckenbaugh	243	Virginia Travers	1
Terence H. Dineen	385	M.W. Luppi	273	David Turner	459
J. Christopher Dodd	225	M.W. Luppi	399	N. Vasudevan	243
L.J. Dotterer	243	Ken J. McDonell	407	Smokey Wallace	117
Andrew Draskoy	111	Richard McGowan	323	Mary S. Weeks	131
Kevin Enderby	347	Alan McIvor	437	M. Wish	159
Harry Forsdick	1	A.S. Melamed	273	Peter Wu	470
Susan A. Funk	95	R.E. Menninger	399	Geoffrey L. Wyant	385
Douglas W. Gilmore	225	Barton P. Miller	470	Michael W. Young	53
Robert A. Gingell	81	Nathaniel W. Mishkin	385	Michael W. Young	185
Robert A. Gingell	131	Joseph P. Moran	81	K. Yueh	273

The Diamond Multimedia Editor

Terrence Crowley
Harry Forsdick
Matt Landau
Virginia Travers
BBN Laboratories, Inc.

Abstract

This paper provides an overview of the Diamond multimedia editor and describes some of the important design and implementation decisions. The editor supports the ability to create documents which include structured, multi-font text, bitmap images, geometric graphics, spreadsheets (and graphics derived from spreadsheets), and digitized voice. All objects are always fully editable and are edited on a single view surface within a single process. Special attention is given to describing the data structures used to support multi-font, multi-style, WYSIWYG text editing for large documents.

1. What is a multimedia document?

A multimedia document is a structured object which contains a collection of objects of different *media types*. Each media has special presentation, representation and editing characteristics. The Diamond editor currently supports five different media types: text, bitmap images, structured graphics, spreadsheets (and graphics derived from spreadsheets), and digitized speech (see Figure 1). We are currently adding other media types to support equations and forms. Each of these media types have special ways of representing their presence in a document and special facilities for editing the content of an object of that type.

2. Focus of Diamond

The Diamond project has focused on the development of facilities to enrich the underlying support for interpersonal electronic communication. Our initial involvement in the area of multimedia documents [FORS84, THOM85] was from the point of view of mail systems rather than technical publishing systems. In an electronic mail system, documents are composed, transmitted, and viewed, more often than they are read in printed form.

This focus had some interesting effects on the view of documents which we provide to the user. For example, in a document which is being viewed electronically, both during composition and after distribution, page breaks are a distraction; within the Diamond editor explicit page breaks are not displayed. In a paper document, a table is a static matrix of labels and numbers, while electronically there may be a rich computational model behind the table. Diamond provides a full spreadsheet editor which allows the recipient of a document to explore the model and not merely peruse the final values of cells in the model.

One of our main goals has been to make the use of graphics in electronic communication as common as the use of text. Five years ago the only graphical facilities widely available were programs which generated output for hardcopy devices or scarce graphics terminals. The advent of the personal computer has made tools with graphical interfaces and which produce graphical output highly available to

the average user, but typically only for a user working alone and producing hardcopy. By providing an integrated facility for displaying graphical output on the screen, on the page, in a complex document or in mail to a colleague, we can make it possible for a user interactively composing a message or an application writer specifying the output of a program to take full advantage of graphical media.

Despite starting with a model of electronic mail rather than technical publishing, we experienced a convergent evolution towards some of the facilities provided in sophisticated publishing applications. There were a couple reasons for this:

1. There was, and continues to be, a need for applications which allow users to easily incorporate graphical objects in their documents. Once Diamond accomplished this for their electronic communications, users desired the same facilities for paper documents.
2. Electronic mail is just one point on a spectrum of communication channels. Any system which expects to play an important communication role must support a range of needs from paper and electronic mail to asynchronous and synchronous conferencing [FORS85].

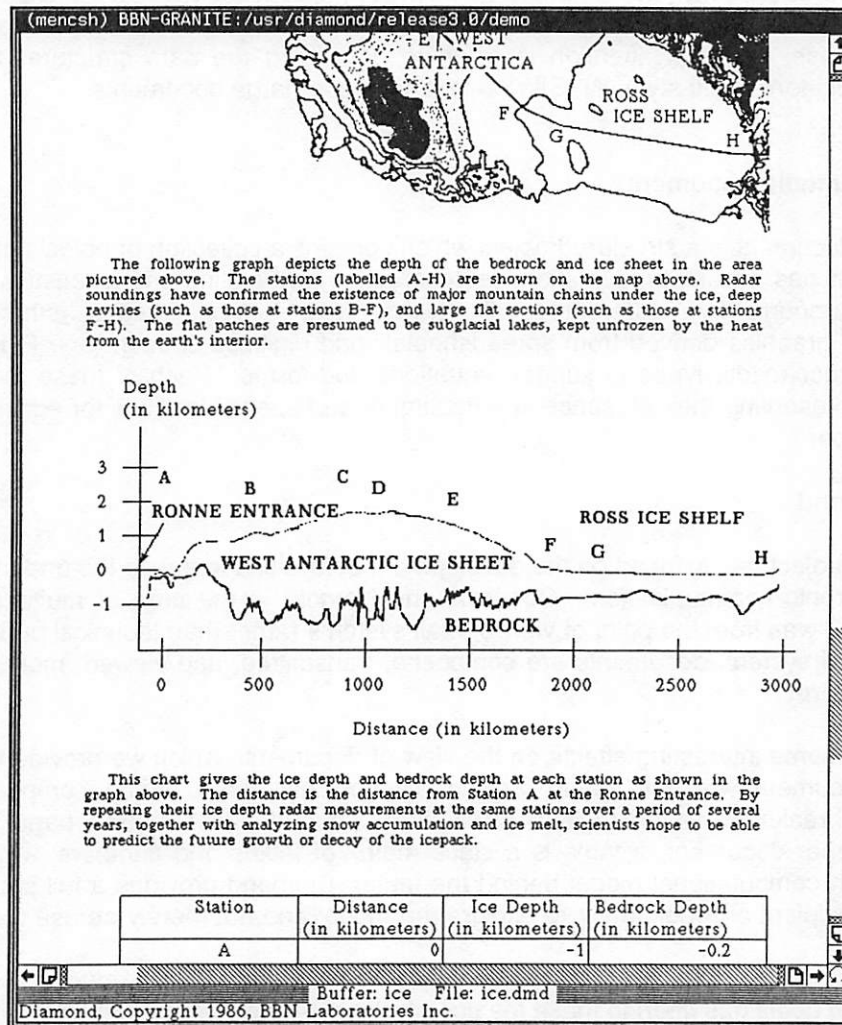


Figure 1: Display of a Diamond Document

3. Editor Design Issues

Because of the limited space available, this paper focuses on the overall editing framework which allows the different media types to interact in a single document, and on the text data structures which form the glue for the other media types. The graphics, image, spreadsheet, and voice modules are powerful and complex editors in their own right. Although the specific issues related to their implementation are not discussed here, many of the generic issues related to their integration into the multimedia editor framework are presented.

There are several key design issues which shaped the form of the editor.

3.1 Document Structure

Diamond documents are hierarchical structures. A hierarchy models the logical structure of documents, *e.g.*, chapters, sections, subsections, lists and sublists. While some document compilers (*e.g.*, troff [KERN76]) allow the user to create documents which appear hierarchical while internally maintaining a flat document model, we felt it important that the implementation represent the hierarchy internally. The internal representation of the hierarchy provides natural handles within the implementation for operating on the logical structure, *e.g.*, selecting an entire chapter or list.

3.2 Media Editor Integration

A number of design decisions revolve around the issue of strong vs. weak integration of the individual editors which make up the composite system. There are several different issues regarding integration:

1. *The display surface.* Are all objects viewable in their "natural" form on a single integrated display surface or are they seen as disjoint objects in separate windows or displays? If they can be viewed in an integrated fashion, are they also edited on that surface or is a separate window used for editing?
2. *Processes.* Are the individual editors run as separate processes or within a single monolithic process?
3. *The user interface.* How consistent are the media editors (in appearance of menus and dialogues, terminology for commands and prompts)?
4. *The data level.* How easy is it to transfer data from one media type to another?

We chose to implement a strongly integrated editor. All objects are displayed and edited on a single display surface and within a single process. The user can switch between editors by simply positioning the mouse over an object and pressing the mouse button to select a feature or pop-up a menu. There are several consequences of this decision:

1. Switching between media types is simple and fast. There is no delay in moving from editing the cell in a spreadsheet, to modifying some text which comments on the spreadsheet, to annotating the graphical object which was derived from the data.

Our experience indicates that the capabilities of a multimedia system will be fully exercised only if the user "cost" of including an object of another media type is kept low. Cost is affected by disruption of context, delay in response, and complexity in the interface. If adding another object means that the object becomes frozen and uneditable within the document, then that object will be added only at the last moment (before sending or printing the document), if at all. If editing that object involves opening up another window on the display and starting another process to

edit the object then small changes become painful and are avoided.

2. The user interface is more consistent. Having all the media editors use the same underlying facilities for menus, dialogues and prompts ensures an important level of consistency.
3. More work is required to implement the multimedia editor. We needed to implement five full-featured editors (as well as the overall structure) instead of being able to directly incorporate existing editors.
4. The editor is large. The executable is currently a little over a megabyte on a Sun3.

While we wanted to strongly integrate the basic media editors, there is certainly a role for less closely integrated facilities. No system can provide all the tools for editing the kinds of data a user might want to include in a document. Other levels of integration might be:

1. Provide an object type which may contain an arbitrary byte sequence and only defines procedures to read and write the data to a file. The presence of the object in the document might be indicated by an icon and some short comment describing its contents and its size.
2. Provide an object type which may contain an arbitrary byte sequence and the name of the program to use for editing it. The multimedia editor could provide mechanisms for running the program in a separate window and passing the data back and forth from the editor to the program.
3. Providing a minimal media editor which provides some representative display of the object within the multimedia editor, but which invokes a separate program to do the actual editing.

3.3 Multiple Buffers, Multiple Panes

There is a need to allow a single instance of the editor to manage multiple documents at one time. In Diamond we provide users with the ability to use multiple *buffers* within the editor. Even in a windowed environment (where multiple instances of the editor can be run), multiple buffers provide a useful mechanism for managing access to a collection of documents.

In addition to the need for managing multiple documents in one editing session, there is a need to see parts of multiple documents (or multiple parts of the same document) at the same time. Multiple *panes* allow a single window (allocated by a higher level window manager) to be split into multiple views. (We use the terms *pane* and *viewport* interchangeably in this paper.) Each view may display a separate buffer or different parts of the same buffer. Panes are useful as lightweight devices for allocating screen space without suffering the overhead of going through the window manager. Their existence is often short-lived as a user splits the view to compare two different documents or reference another part of the same document.

3.4 Large Documents

The document editor is designed to handle large documents (hundreds of pages). We use the editor for long reports, proposals, and manuals, as well as short spontaneous messages. It is important that the internal data structures and algorithms efficiently handle large documents. As an example of how this requirement affects our design, it became clear that objects could not have an absolute position within the document (as was the case in earlier versions of the editor). Absolute positioning would mean that changes early in the document affect everything after them; this was an unacceptable performance penalty. The current system defines a layout algorithm which can proceed from any object and layout the adjacent objects above and below a distinguished object. The position of a *viewport* over the document is

based on this object rather than an offset from the top of the document. This topic will be discussed in further detail below.

3.5 External Representation

Where possible we have attempted to use an ASCII external representation. The benefits of a human-readable output form have been catalogued elsewhere, but in general it opens the system to creative interaction with other applications. It makes it possible to use existing text editing tools to examine and modify documents and simplifies the automatic generation of multimedia documents by other programs.

We have also attempted to use standards where they are available. Images are stored using a standard format and we use the Lotus 123 format for storing spreadsheets. We make use of the PostScript page description language for printing, as well as using a "stylized" PostScript as the graphics output format. We are currently examining the use of the Office Document Architecture (an ISO Draft International Standard) for representing the document structure [ODA86].

4. Editor Framework

The editor framework provides facilities for managing multiple buffers and panes and dispatching events to the individual media editors. It provides functions which operate on entire objects and entire documents, as well as a set of services which are used by the media editors. Each media editor makes available an array of generic functions which are called by the editor framework, as well as defining whatever media specific editing operations are appropriate.

4.1 Editor Services

Besides the modules which manage multiple buffers and panes and operations on documents as a whole, there are a number of services provided by the framework which are used by the individual media editors.

4.1.1 Event Dispatch

The editor framework handles the task of dispatching input events to the individual objects. There is a single command loop which looks to see if any characters are waiting on the event queue. If none are available, it checks to see if any of the asynchronous activities like displaying the screen or highlighting the selected region need to be processed. When a character arrives, the main loop examines the character to determine if it was a mouse button event. If it was, the current pane and object are first updated to correspond to the pane and object underneath the mouse. Processing the mouse event then proceeds as with any normal character.

At any time there is a current *keymap*. The keymap maps character codes onto editing functions. When an object is selected, (*e.g.*, in response to a mouse event over it) it makes its keymap the current one. The keymap will typically contain some functions which are common to all the media editors (*e.g.*, Control-V bound to page-down) as well as functions which are specific to the particular media type being edited. Functions installed in the keymap take no arguments but examine a few globally accessible variables to determine the context in which they should be executed (the current document, the current selection). Menus are handled simply by binding a function which pops up a menu to one of the mouse buttons. Menu descriptions contain the command name as well as a pointer to the function which should be executed when a particular command is selected. These are the same functions which are bound to keyboard commands.

4.1.2 Shared Table Maintenance

Certain information needs to be specified or shared by multiple objects across a single document. This information is specified in a variety of *tables* associated with a document. Entries in tables are normally accessed by an index into the table, though some tables support named entries. Tables perform two functions:

1. They bundle together information which needs to be specified by an object so it may be referenced more efficiently, *e.g.* a font change in the middle of a text passage is stored as an index into the font table rather than requiring the font family, face, and size to be specified "in line".
2. They also gather together data which needs to be shared between objects, *e.g.*, text formatting styles or graphical fill patterns.

There are six kinds of tables: color, font family, font delta, textures, line styles, and text formatting styles. The color table defines RGB color values. The font family table defines the family of fonts used in the document (*e.g.*, Times, Helvetica, *etc.*). The font delta table defines incremental changes to the font of a passage (the font model will be described in more detail below). The texture table defines a set of fill patterns. The line style table bundles line attributes such as line width, pen fill texture, and pen shape. The text style table bundles text formatting attributes together into named styles.

The principal generic issues in table maintenance are merging tables when inserting one document into another, the associated task of updating table references in the inserted objects to point into the new tables, and deleting unused entries on output.

4.1.4 Clipboard Management

The clipboard supports cutting and pasting within a buffer, between buffers managed by the same process, and between multiple editor processes. The clipboard is implemented as three files. The first file records the process and buffer which last wrote to the clipboard. The next file records the document table information. The final file contains the actual data which was copied to the clipboard. The concatenation of the table file and the data file is always a legal multimedia document (as opposed to having a special clipboard file format for partial objects). The tables and data are stored separately because reading the tables and merging them into an existing buffer is a relatively expensive operation. For the common case where the cutting and pasting happens within the same buffer, there is no need to read the tables file if the tables have not changed since they were last written.

4.1.5 User Interface Routines

The user interface routines provide a number of functions for gathering menu and form-based input. The menu system provides hierarchical, context-sensitive menus.

The forms system provides a flexible interface to command buttons, choices, boolean check boxes, and type-in fields (see Figure 2).

SpreadSheet Formats

Format Style:

☒ General ☐ 1234

☐ 1234.23 ☐ 1,234

☐ 1,234.23 ☐ \$1234

☐ \$1234.23 ☐ \$1,234

☐ \$1,234.23 ☐ \$123.4

☐ A1 + B2

☐ Precision Digits:

OK Cancel Keep Up

Figure 2: A form.

One interesting user interface technique addresses the problem of allocating screen space in an application with five full-featured editors. At times, pop-up forms and menus can become awkward when a user needs to carry out some operation repeatedly. Many stand-alone applications address this issue by placing command "buttons" which always remain visible along the edge of the screen. Providing all the command options in static forms would not only tax the user's visual attention, but also use up all of the available screen space. To address the issue, many pop-up forms have a "Keep Up" option which will detach it from the current window and place it in a separate window. The form then remains visible and may be moved around like any other window on the screen. For example, in Figure 2 above, if the *Keep Up* button is selected then the pop-up form will be put into a window of it's own along side the editor's display window.

When an event occurs within a window containing a form, it is processed by the forms interface. The selection of a command button in a detached form (as in a pop-up form) causes the invocation of the associated editing procedure. This procedure examines the current document and object to determine the context in which it was invoked.

4.2 Generic Media Functions

Each media editor defines an array of generic functions which may be called by the editor framework. These include routines to create a new object, destroy an object, read and write an object to a file, change the size of an object, display it on the screen, generate output for printing, respond to selection or de-selection, and a few other miscellaneous functions.

The definition of this set of generic functions provides the basis for extending Diamond. The implementor of a new media type is required to provide an implementation of the well-defined generic functions. That implementation can draw on a variety of support routines that are part of the editor framework.

4.3 Media-Specific Editing Functions

The generic functions provide the core implementation of a media type. However, they do not address the issue of editing the *content* of an object. Each media type defines a number of *media-specific* editing functions. When an object is selected, it installs a keymap containing pointers to the media-specific functions, which allows them to be invoked through keyboard commands or menu selection.

Besides publicizing the functions through the keymap, a media editor also assigns a name to each editing function to allow it to be referenced symbolically. This is used to provide a facility for rebinding

functions to keys and menus and for displaying the current keystroke to function mapping.

5. Display

Updating the display is one of the most complex areas of the editor. The activities of determining the position of the objects in the viewport (*layout*), interactively formatting, painting their contents, and responding to user actions all interact. Figure 3 provides a flowchart of the processes involved. The main command loop dispatches tasks. If user events are pending, they are processed first. Once all events have been processed, the layout is determined; this may involve reformatting. Once the layout is correct, any objects queued for painting are drawn; this may also require reformatting.

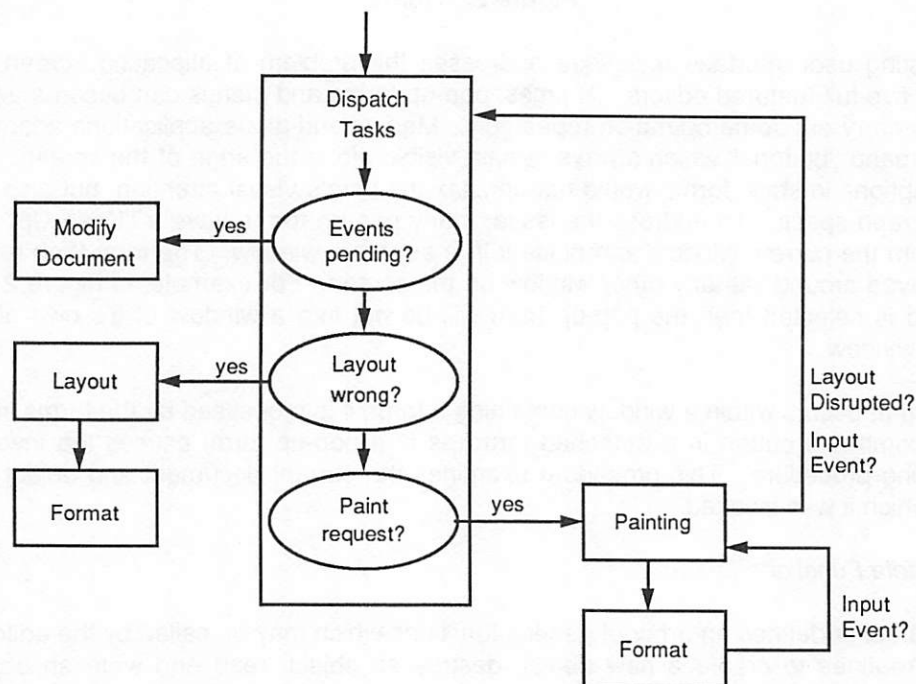


Figure 3: Interaction of Editor Input Event, Layout, Painting, and Formatting Algorithms

If an event is received while an object is being painted, the act of drawing the object is interrupted and the pending events are processed immediately. Operations which modify the content of the document do not directly update the display, but rather mark areas of the document as having changed and possibly needing redisplay.

The notion of asynchronous display update is a familiar one in full screen editors [STAL81, HAMM81, ALLE81, FURU82]. To summarize, there are a couple reasons why this is a critical design decision:

1. *Subsequent editing operations may have made the current display activity unnecessary.* The standard case is a series of page-down commands. As the first page is being displayed, the next page-down command makes it unnecessary to finish the display of the first page. The ultimate effect of interrupting the display to process user actions is that the editor appears more responsive to the user.
2. *Every editing operation is not required to know how to update the display.* Most operations can be defined as a composition of more basic operations. Since the basic operations must handle

the bookkeeping of display update (because they may be invoked directly as a single operation), the composite functions can ignore the issue of display update entirely. A composite operation does not have to worry about the unusual display effects of combining more basic operations since it knows the display will be updated in the most efficient manner possible after all the more basic operations have been processed. For example, the operation to globally replace a text string can make use of the more basic search, insertion and deletion commands without worrying about how the display should be modified.

The value of this design decision cannot be overemphasized. It permeates the implementation. While this is a familiar feature in full screen text editors, it is somewhat more unusual in a graphical application such as our multimedia editor. In addition, the implementation is quite different in a multimedia editor than in a text-only editor.

There are three separate activities involved in updating the display. The *layout process* determines the location of objects on the display. The *formatting process* determines the appearance of an object. The *painting process* actually draws an object on the screen.

5.1 Layout

On the screen, the document is layed out in a vertical galley with no pagination. The layout of objects within a pane needs to be recalculated when:

- o a new buffer is displayed,
- o the *viewport* (the portion of the document which is visible in a pane) changes,
- o objects are inserted into or deleted from the buffer,
- o objects change size.

The *viewport* is specified by a distinguished object displayed in the pane and the offset of the top-left corner of that object from the top-left corner of the pane (see Figure 4). A critical feature of the layout algorithm is that it is possible to start from any object and proceed to layout the objects which appear in the viewport above and below it with a bounded amount of work (*i.e.*, we will not have to proceed too far backwards or forwards in the document before we can determine that no other objects may overlap the viewport). While that may seem like a trivial criteria, it needs to be reconciled with the ability to place objects as the user would like.

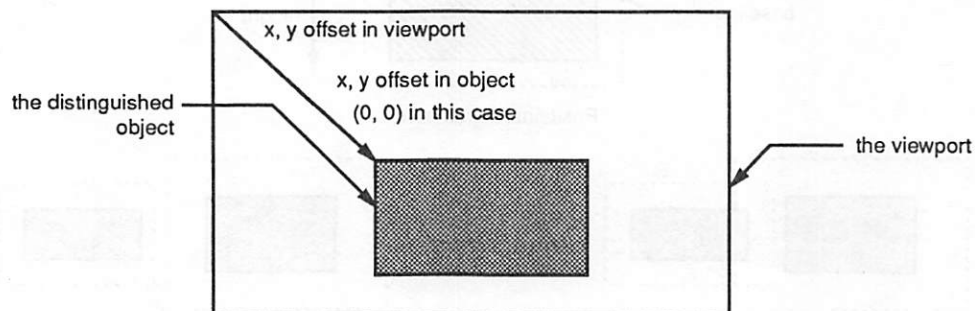


Figure 4: Layout of objects in viewport.

5.1.1 Layout Requirements

In a vertical galley, the algorithm which would allow the most precise control over the position of objects would be to simply allow a user to place any object at an absolute offset on the galley. This would allow any object to be placed in any desired juxtaposition with another. Most "draw" style graphical editors provide this type of positioning control.

For positioning objects in a document, this simple facility has several problems.

1. It does not capture the structural model behind the document which allows us to compute algorithmically the relative position of objects. For example, because of the structure of the document we know that one paragraph should lie above another or that the tag on an enumerated paragraph should lie to its left.
2. The position of objects needs to be updated when new objects are inserted or deleted.
3. It makes it difficult to determine which objects are visible in the viewport without scanning the entire document (which is the algorithm used in most graphics editors, but inappropriate for a document editor managing many objects).

However, there are times when the ability to position objects relative to one another is desirable, *e.g.*, when positioning a drawing alongside a spreadsheet or a paragraph. The layout algorithm needs to solve the problems described above as well as the desire to position objects under the user's control.

5.1.2 Layout Algorithm

Each object has a width and height, a vertical and horizontal offset from some nominal origin, a surrounding margin area, and a baseline (see Figure 5). To determine the layout of objects in a pane, we start from the distinguished object and gather objects into *frames*. Consecutive objects lie in the same frame if they do not *conflict*. Objects conflict if they would overlap when placed relative to the same nominal origin or if the margin of one object overlaps another object. The margins of two objects may overlap without inducing conflict.

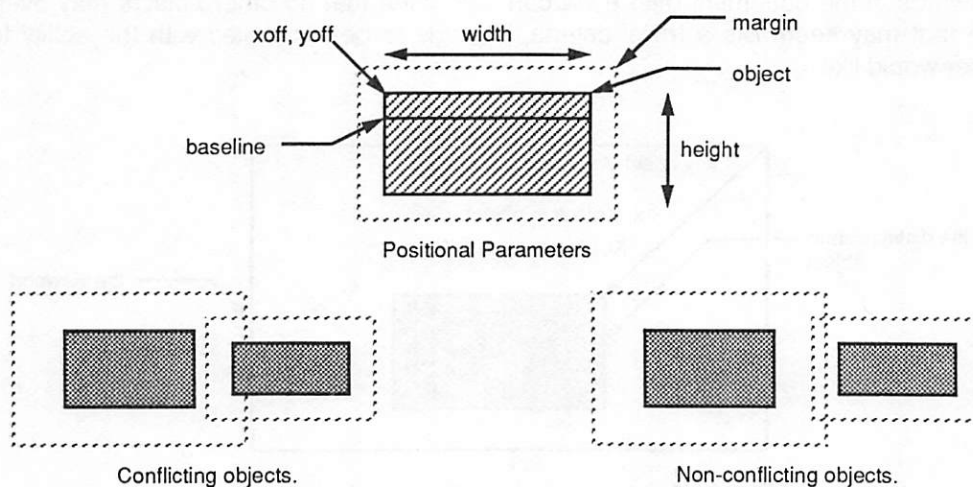


Figure 5: Frames.

By assigning appropriate offsets so that their positions do not conflict, we can place a set of objects in a frame in arbitrary relation to each other. The positional attributes of text objects are normally derived automatically during the formatting process (based on the margins, pagewidth, font height, *etc.*). Other objects (such as a spreadsheet or image) are assigned initial positions, but the user interface of the editor allows the user to adjust these, *e.g.*, to place a graphics object along side a spreadsheet.

Objects in a frame are merged to form a composite object and margin. Frames are layed out so that the margin below one frame overlaps the margin above the next without conflicting.

5.1.3 Layout Process

During the first step of the layout process, we group objects into frames and lay out frames forwards from the distinguished object until we run out of space in the viewport below it and lay out frames backwards from the distinguished object until we run out of space above it. Because of the way frames are grafted together we know we can halt the layout process once a frame lies partially outside the viewport. At the conclusion of this step we record the positions of the visible objects in the viewport. We also set the distinguished object to be the first object visible in the viewport (the distinguished object which drove the initial layout may not even be visible).

The second step of the layout process is to install the new positions as the current layout of the pane. The installation process has the goal of minimizing repainting of objects by copying the old image of an object from one place on the display to another (if possible) and of aggregating the copying operations (since the cost of moving an image on the display is much more sensitive to the number of move operations than to the size of the region moved). In most cases the moves can be aggregated into a single operation. Note that this analysis is done at the level of complete objects such as paragraphs or an entire graphical drawing and not at the level of individual lines of text or other graphical features. A typical viewport will only contain from ten to fifteen objects. A complex one might contain thirty.

Many operations degenerate to selecting a new object to be the distinguished object in the viewport (or changing the offset at which the current distinguished object is to be displayed) and then requesting the viewport to be re-layed out. For example, scrolling operations simply involve changing the offset of the distinguished object in the viewport. It is up to the layout process to redraw the screen in the most efficient way possible. An object is centered in the pane by making it the distinguished object and then setting the vertical offset to place it in the middle of the viewport. At the end of the layout process the distinguished object will be set to the first object displayed in the viewport.

5.2 Painting

Once the layout is determined, requests are generated to paint an area of an object. Requests can be generated in two ways: external to an object or internal to an object. An *external* request occurs when an object is inserted into the document or scrolls into the viewport. In these cases, the framework routines have determined that a region of the object that is not already on the screen needs to be painted. An *internal* request is generated when an object is modified through the media-specific editing routines, *e.g.*, the fill pattern of a circle in a graphical object changes. A queue of these outstanding paint requests is maintained and serviced for each pane.

Each object is made up of a set of features which are drawn atomically. It may be a line of text, a row in a spreadsheet, or a rectangular region of a bitmap image. The individual media editor determines the appropriate granularity for their application. When a paint request is generated internally by a media editor, it marks which of these atomic display features need to be repainted (are "*dirty*") and then queues a request with the framework routines to indicate that the object needs repainting. When a paint request

is generated externally, the framework routines call one of the generic media functions to mark the area of the affected object. The media-specific routine determines which features lie in that region and marks them dirty.

During the display cycle, the outstanding paint requests are processed in order. The first step in processing the request is to determine if the object named in the request is still visible in the pane (since the layout may have changed since the request was queued). If it is visible, the media-specific paint procedure is called and proceeds to draw the features which have been marked dirty. Between drawing each atomic display feature, it checks for outstanding input events. If there are outstanding events, the paint procedure is interrupted and the events are processed. If the paint procedure is not interrupted, it returns normally and the request is removed from the queue.

5.3 Multiple Panes

Multiple panes do not affect the layout process greatly. Since each viewport is defined by a pointer into the document rather than an absolute offset from the top of the document, changes to a document in one pane do not necessarily affect other views of the document in other panes even if large sections of the document "above" the viewport were modified. When an editing action which may cause the layout to be affected (*e.g.*, objects are deleted) is invoked in one pane, the layout is recalculated for every pane displaying the document. Since normally the viewport of the two panes do not overlap, this does not usually affect the display in the other panes.

Multiple panes do add an additional complexity to the painting process, since the same object may be visible in different panes. A separate paint queue is maintained for each pane. The "dirty" bit which is maintained by the individual media editors for each atomic display feature is actually a *dirty set*, with a separate bit for each pane. This adds a further distinction between external and internal paint requests. On an external request, features are marked dirty and queued for a single pane; on an internal request (generated because of a change to the content of the object), features are marked dirty for *all* panes and a request is queued for all panes in which the object is visible.

5.4 Formatting

The discussion of layout and painting has ignored the issue of incremental formatting. While the hooks are in place to allow any media to do incremental formatting, text is the only medium which is currently performing any significant formatting so this discussion will be motivated around text.

The text editing routines provide a WYSIWYG interface. As characters are inserted or deleted, or formatting attributes are modified, the display is updated to reflect these changes. When a change is made to the text which affects the formatting, the modified region is marked as *damaged*. For each text object in the damaged region, the line at which the damage started is recorded; reformatting will begin at this line (actually the one above it, since a change on one line can affect one line above). A request to paint each of the damaged objects is placed on the paint queue.

There are two places where the text might be reformatted (see Figure 3). When called to paint an object, the text routines first format the paragraph, beginning at the line where the damage started. If formatting caused the object to change size, the framework is informed and the viewport must be re-laid out. If an input event arrives during the formatting process, formatting is abandoned, but rather than immediately returning control to the event handler, at least two lines of text are painted. This ensures that even if a user is typing extremely fast a couple lines of context will remain accurate. When the user pauses, the rest of the paragraph is reformatted and redisplayed.

The second place where a text object is formatted is during the layout process as objects are gathered into frames. Strictly, this is unnecessary, since objects will be formatted during the painting process. However, when an object changes size during the painting process, painting is abandoned and the layout is recalculated immediately (assuming there are no waiting input events). If we had twenty paragraphs in our viewport, all of which needed to be reformatted and change size (e.g. because the global document font size had been changed), and we waited to reformat during the painting process, we would end up laying out the pane twenty times, as each paragraph was displayed. This is not only inefficient, but also produces an unpleasant visual effect. Note that the layout process only needs to be invoked when the paragraph changes size, not during small editing changes which have no effect on the overall size.

6. Text Internals

Text occupies a special role within the editor. Most media-specific operations affect the contents of a single object. Text operations routinely span multiple objects, e.g., when two paragraphs are selected and deleted, or when the cursor moves from one paragraph to the next. The text data structures also provide the framework for anchoring the object hierarchy and the text routines provide the user interface to the structured document facilities.

The data associated with most objects resides at the leaves of the document hierarchy. For text, there is a single data structure which contains the characters for all the text objects in the document. There are several reasons why the implementation was organized in this way.

1. *Text operations span multiple objects.* These operations have simpler implementations when there is a single coordinate space for indexing into the character buffer (rather than requiring a pointer to the object and an index into it). This single coordinate space also makes it trivial to determine the relative ordering of two pointers into the document.
2. *Many text objects are small.* Using a shared data structure pro-rates the data structure overhead across all of the objects.
3. *The text data structures provide a linear space for anchoring the objects of the hierarchy and describing regions of the document.* All media objects (whether text or graphical) contain a pointer into the linear character buffer. An auxiliary structure allows us to go from a position in the linear buffer to the corresponding object in the hierarchy. This allows us to map back and forth between the object hierarchy tree and the linear position of objects in the document without walking the tree.

6.1 Character Buffer

The character buffer is implemented as a single gapped buffer (see Figure 7). This data structure has the property that insertions and deletions at the locus of operation are essentially free. When the locus moves, characters must be copied across the gap. When the gap is filled, a new buffer must be allocated and the characters copied to the new buffer. The cost of these major copying operations is endured for the efficiency benefits of the more typical situation where many insertions and deletions occur in close proximity. In addition, functions which operate on the characters in the buffer can almost treat it as a single sequence of characters (with care on stepping around the gap) so their implementation is simplified.

On insertion, characters are copied into the gap and the gap pointer and character count are incremented. Deleting backwards simply involves decrementing the gap and the character count; deleting forwards is done by decrementing the character count.

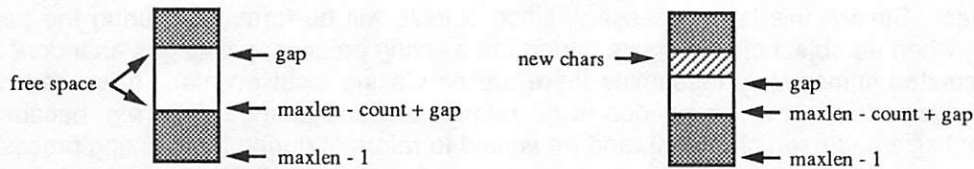


Figure 7: Gapped character buffer.

6.2 Markers

A *marker* is a robust pointer into the character buffer which remains valid in the face of insertions and deletions in the buffer. The characters in the buffer can be viewed in two coordinate spaces. One space is the *real* space which runs from zero to maxlen. The other is the *virtual* space which runs from zero to the character count. We treat an index into the buffer (whether real or virtual) as a pointer *between* two characters rather than as a pointer *to* a character. This removes any ambiguity when discussing insertion or deletion at an index or having to deal with special cases of indices at the beginning and end of the buffer.

A marker is an index into the *real* coordinate space. The key to this representation is that indices only need to be updated when characters are copied across the gap, and only those markers which pointed between the copied characters need to be updated. The same updating patterns which make the gapped buffer an efficient representation for characters makes the marker an efficient representation for pointers into the document. Markers are allocated for a variety of purposes. The allocated marker is passed to whichever function requested it and is also recorded in a sorted array so its value can be updated when the gap of the buffer changes. The sorted array of markers is also implemented as a gapped buffer, since the pattern of allocation of markers follows the same pattern as the operations on characters in the character buffer.

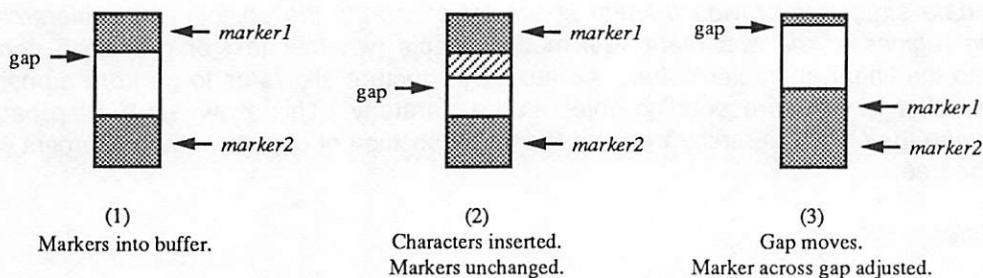


Figure 8: Updating markers.

6.3 Formatting Information

The format of a text passage is specified in a manner similar to Scribe[REID80]. Named styles are defined with a set of attributes such as font, left and right margins, lineheight, justification, *etc.* Styles may be defined with inheritance, so, *e.g.*, I can define a "Quotation" style which is just like "Paragraph" except the left and right margins are moved in a half inch. If the Paragraph style is redefined, the Quotation style will inherit the new values of those changed attributes which are not locally overridden.

Each text object or paragraph (we use the term paragraph in a generic sense to mean some block of

text) has a pointer to the style which is used to format it. If the style is altered, the text object will be reformatted. Each paragraph also has a marker to the location of the end of the paragraph in the character buffer. This marker points before a special character in the buffer which tags the end of the paragraph. The presence of this character in the buffer enables us to distinguish a marker which points after the last character in one paragraph from a marker which points before the first character of the next.

Within a paragraph, formatting changes can occur at the character level. These changes are recorded in a single document-wide format change table as markers with associated formatting attributes. The format change table is also implemented as a gapped buffer (for the same reasons as the character buffer) with entries stored in sorted order based on the value of the associated marker.

For character-level formatting changes, we store the location of the start of the change and the end of the change. Within the data structures, formatting changes may strictly nest but may not partially overlap (since the end format change mark does not explicitly indicate the formatting change it is ending, it is always presumed to be the most recent unmatched one). Format changes also may not cross paragraph boundaries. When the user attempts an operation which would violate these rules (*e.g.*, selecting two paragraphs and changing the font face to italic), formatting changes are inserted for each paragraph in the selected region. By enforcing this restriction, we bound the amount of work which is required to determine the format of a particular line of text, since we only have to search as far back as the beginning of the paragraph to find any incremental formatting changes which apply at a particular point.

Font changes are stored as incremental deltas to the current font. For example, we can say that the next character is displayed in an italic face without having to specify the family or size. The advantage of this representation over one where an absolute font is recorded at each change is that an operation like changing the font family used to display the entire document is trivial, since no work has to be done to update the incremental font changes.

7. Structural Issues

The document editor provides the facility for creating hierarchically structured documents. The external (leaf) nodes of the tree are instances of the Diamond multimedia object types: text, bitmap images, geometric graphics, spreadsheets, and digitized voice. The leaf nodes are linked together by *list* nodes into a hierarchical structure. As described above, each leaf node contains a marker into the linear character buffer. For a text object, this marker indicates the end of the paragraph and all the characters up to the previous marker lie in the paragraph. For other objects, the marker points before a single character and the rest of the object data is stored at the leaf node.

Most of the attributes of list nodes are due to their use to structure text, although list nodes are used generally to organize all of the media types supported by the editor. A list format has a name and a number of attributes which affect the formatting of the multimedia object elements within the list. Lists may be nested. In addition, a *tag* format is specified as part of the list format and indicates a text object which should be automatically inserted into the document before any element in the list. A *label generating* string may be specified which indicates how the contents of the text tag should be generated based on its location in the hierarchy (*e.g.*, to automatically number the items).

As an example, the default "Enumeration" list format specifies that all objects should be indented one-half inch and that tags should be generated using the "Itemtag" format and with a label generating string which specifies Arabic numbering.

A number of issues arise regarding the implementation of hierarchical structuring:

1. *Is there any reflection of the hierarchical list structure in the linear character buffer? For example, are there special characters marking the beginning and end of a list?*

The current implementation places no special characters in the buffer to mark the list structure. The document structuring is only reflected in the object hierarchy. The special characters would have no significance to most of the character editing and movement commands and so would have to be handled in a special manner. The complications this would have introduced into the text operations were deemed large enough to warrant not having a direct representation in the linear text structure.

One problem with this choice is that it makes it impossible to use markers into the text buffer to unambiguously specify a location in the hierarchy. For example, there is no way, using markers, to disambiguate the position after the last item in a list and before the next item outside the list. Although in the current interface there is no way for the user to make this distinction, internally there are places where such a facility would be useful. The consequence is that there are a number of *ad hoc* techniques for addressing the issue when the distinction is meaningful.

2. *How is the structural integrity of the document maintained (i.e., how are tags generated and how is automatic numbering of lists implemented)?*

The maintenance of the structural integrity is an asynchronous activity similar to the display process. When objects are deleted from or inserted into the object hierarchy, or when levels of the hierarchy are added or removed, the affected areas of the document are marked as being *list damaged*. When there are no input events to process, the list damaged regions are examined to ensure that all list elements have an associated tag item and that all automatically generated tags are embedded within lists. At this time, the content of the tags are also examined to guarantee that they correspond to the values specified by the label generating string.

3. *What kinds of constraints are there on selection of regions crossing structural boundaries?*

The editor does not allow a selection region to contain a partial sub-tree if the beginning or end of the region extends beyond the sub-tree. For example, in Figure 9 we cannot select from outside an enumeration into the first two paragraphs of the enumeration (leaving subsequent enumerated paragraphs unselected). The main motivation behind enforcing this constraint is the desire to have the cut and paste operations remain symmetrical.

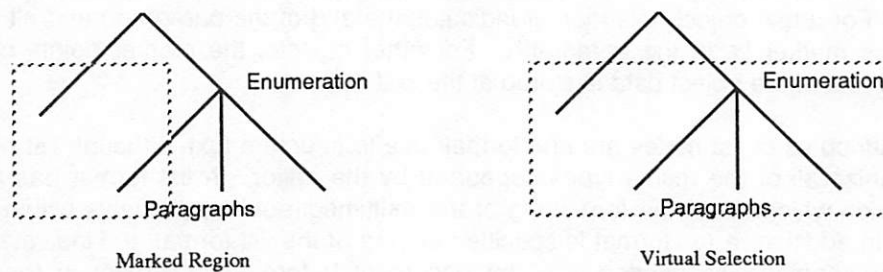


Figure 9: Constraints on Selection

If we allow the selection of a partial sub-tree, then we have a difficult issue dealing with the internal node of the sub-tree. Presumably the cut buffer would record that the two paragraphs were embedded in the enumeration. However, the entire enumeration was not cut, so it also remains within the document. If we were to paste the cut buffer back in, we would end up with two separate enumerations. By constraining the semantics of selection, we finesse this problem away. From the user's point of view the effect is minimal, since that type of selection is unusual

(because of the semantic basis for the existence of the structure in the first place).

8. Other Media

The other media types (graphics, images, spreadsheets, voice) have received summary treatment in this discussion, but, in fact they are no poor cousins. The same goals which led us to provide sophisticated facilities in text support drove us to implement full-fledged media editors.

Graphics

The graphics editor provides the ability to create structured graphical objects which may contain lines, boxes, circles, ellipses, arcs, splines, polygons and text. Outlines may be drawn in a range of brush widths and textures, and objects may be filled with a texture or be transparent. Editing aids such as rulers and grids with gravity alignment are also provided. Objects may be grouped, scaled, and rotated.

Spreadsheets

The table editor is a full-fledged electronic spreadsheet compatible with the data format of Lotus 123 (though the user interface is completely different to take advantage of the superior graphics facilities available). A spreadsheet may be 256 columns by 2048 rows. A full complement of functions are provided as well as range of recalculation controls allowing calculation by row, by column, or in "natural" order. Recalculation may iterate a fixed number of times or until some criteria is met. Rulings around cells may be used to give the appearance of a complex table or form.

Graphs may be derived from the data in the spreadsheet and are generated as full-fledged graphical objects which may then be manipulated and annotated with the graphics editor.

Images

The editor currently supports monochrome bitmap images. Images may be cropped, scaled, rotated, reflected, painted on, or marked with graphical annotations. Within the image editor, graphical annotations get merged into the bitmap image. Images may also be included as objects in the graphics editor, and in that context graphical annotations retain their separate nature.

Voice

Voice may be added to a document as an annotation. We are currently using a LPCM-10 vocoder developed at Lincoln laboratories for voice input and output. This device compresses speech to 2400 bits per second. The presence of speech in a document is indicated by a small loudspeaker icon. A user hears the speech passage by selecting the icon with the mouse. Since we view voice as a relatively "spontaneous" medium, we have provided a simple interface to inserting and editing voice, modeled on a handheld tape recorder.

9. Status

Diamond was developed under contracts sponsored by the Defense Advanced Research Projects Agency (DARPA) under Contract No. MDA903-83-C-0131 and by the National Science Foundation as part of the EXPRES project, a pilot project in electronic submission of proposals and cooperative work.

References

- ALLE81 ALLEN, T., NIX, R., AND PERLIS, A. "PEN: A hierarchical document editor." In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices (ACM)* **16**, 6 (June 1981), 74-81.
- FORS84 Forsdick, H. C., Thomas, R. H., Robertson, G. G., Travers, V. M., "Initial Experience with Multimedia Documents in Diamond," In *Computer Message Service, Proc. IFIP 6.5 Working Conference*, IFIP, (1984) Pages 97-112.
- FORS85 Forsdick, H. C., "Explorations into Real-time Multimedia Conferencing," In *Proc. Second International Symposium on Computer Message Systems*, (Sept. 1985)
- FURU82 FURUTA, R., SCOFIELD, J., AND SHAW, A. "Document formatting systems: Survey, concepts, and issues," *Comput. Surveys* **14**, 3 (Sept. 1982)
- HAMM81 HAMMER, M., ILSON, R., ANDERSON, T., GILBERT, E.J., GOOD, M., NIAMIR, B., ROSENSTEIN, L., AND SCHOICHET, S. "The implementation of Etude, an integrated and interactive document production system." In *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices (ACM)* **16**, 6 (June 1981), 137-141.
- KERN76 KERNIGHAN, B.W. "A TROFF tutorial." Internal Memo, Bell Laboratories, Murray Hill, N.J., Aug. 1976.
- ODA86 "Information Processing -- Text and Office Systems -- Office Document Architecture (ODA) and Interchange format -- Parts 1-6", ISO/TC 97, September 1986.
- STAL81 STALLMAN, R. M. "EMACS, the extensible, customizable self-documenting display editor." In *Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation, SIGPLAN Notices (ACM)* **16**, 6 (June 1981), 147-156.
- REID80 REID, B. K., AND WALKER, J. H. *SCRIBE Introductory User's Manual*, 3rd ed., preliminary draft. Unilogic, Pittsburgh, 1980.
- THOM85 Thomas, R. H., Forsdick, H. C., Crowley, T. R., Robertson, G. G., Schaaf, R. W., Tomlinson, R. S., Travers, V. M., "Diamond: A Multimedia Message System built upon a Distributed Architecture", *Computer*, (December, 1985).

An interactive WYSIWYG table editor

S. L. Murrel

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

D. De Baer

University of Antwerp
B-2610 Antwerp, Belgium

Many who routinely employ word processing systems to produce documents continue to manually add tables, pictures and graphs. These special-purpose utilities commonly lag behind more general editing and formatting capabilities. Certainly this has been true within the standard *troff* environment. More and more small languages, e.g. *pic*,¹ *grap*,² are providing new features and functionality. Such programmable tools are essential to the productive and efficient development of yet more tools with yet more features and functionality.

Improved user interfaces are also essential to the introduction of this added functionality to more casual users. Recent times have witnessed improved visual interfaces to various preprocessors. *Cip*,³ for example, provides a bitmap interface to *pic*'s picture capabilities. More such work will enable us to more fully automate document preparation.

1. *Vtbl*

Vtbl, the visual table program presented here, is an interactive 5620 or Blit-based⁴ table editor which provides a WYSIWYG visual interface to *tbl*.⁵ *Tbl* is powerful but difficult to use; *vtbl* makes it easy for occasional users to create complex tables. Tables are composed of columns which may be independently centered, right-justified, left-justified, or aligned by decimal points. Headings may be placed over single columns or groups of columns; labels can be written for a row or spanning several rows; data within the table can freely span columns and/or rows. *Vtbl* reads *tbl* commands to construct tables and turns the modified WYSIWYG display into *tbl* commands.

Vtbl facilitates the editing of 2-dimensional tables by supporting cut, snarf and paste operations on groups of rows, columns and boxes as well as lines, line segments and points. Starting with a single box, an empty n by m table frame, or an existing table, the user builds the table using the mouse, its menus and the keyboard. Columns/rows can be added or exchanged; *vtbl* automatically matches the surrounding context. When a partial column or row is added, adjacent boxes span it.

Individual boxes can be selected from the keyboard, while points, line segments, boxes and groups of boxes can be selected using the mouse. *Vtbl* displays the box frames as dotted lines and indicates selected objects using bold lines. These dotted lines do not affect the final output.

Previously entered data files can be read into *vtbl* or text can be typed into selected boxes from the keyboard, with *vtbl* automatically adjusting the box widths. In addition, the user can force changes in width by moving the line intersections. Using the mouse menus, boxes can be aligned and spanned vertically and horizontally. Spanning a box is as easy as sweeping the mouse over the boxes to be spanned. Withdrawing a spanned box restores the underlying box grid.

Cutting a single box or a group of boxes deletes text and special spanned structures, restoring the column/row frame. Cut and snarf objects can be freely pasted within the table, replicating the existing type and structure information of the columns to the right and the rows beneath. Box groups can be pasted over other box groups or at points or line segments. When pasted at a line segment, adjacent boxes span the new subtable. The entire table can be snarfed and pasted at the

bottom or to the right!

This paper describes how to create tables using *vtbl*. It should make creating a table as easy as sweeping a few rectangles on the 5620.

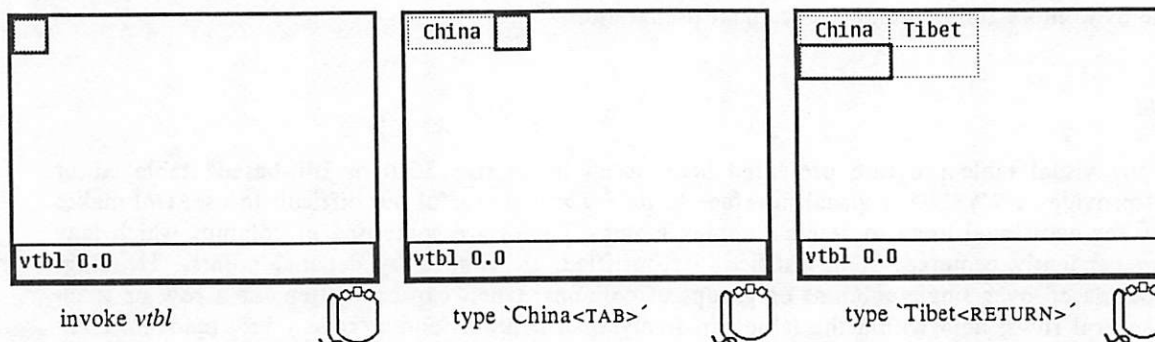
1.1 Designing a table from scratch

You are faced with the problem of getting the table in front of you, which describes trips to China and Tibet, into a manuscript:

China		Tibet	
N	Beijing	S	Lhasa
5 cities		monastery	
6 ports			
everywhere N, S & urban			

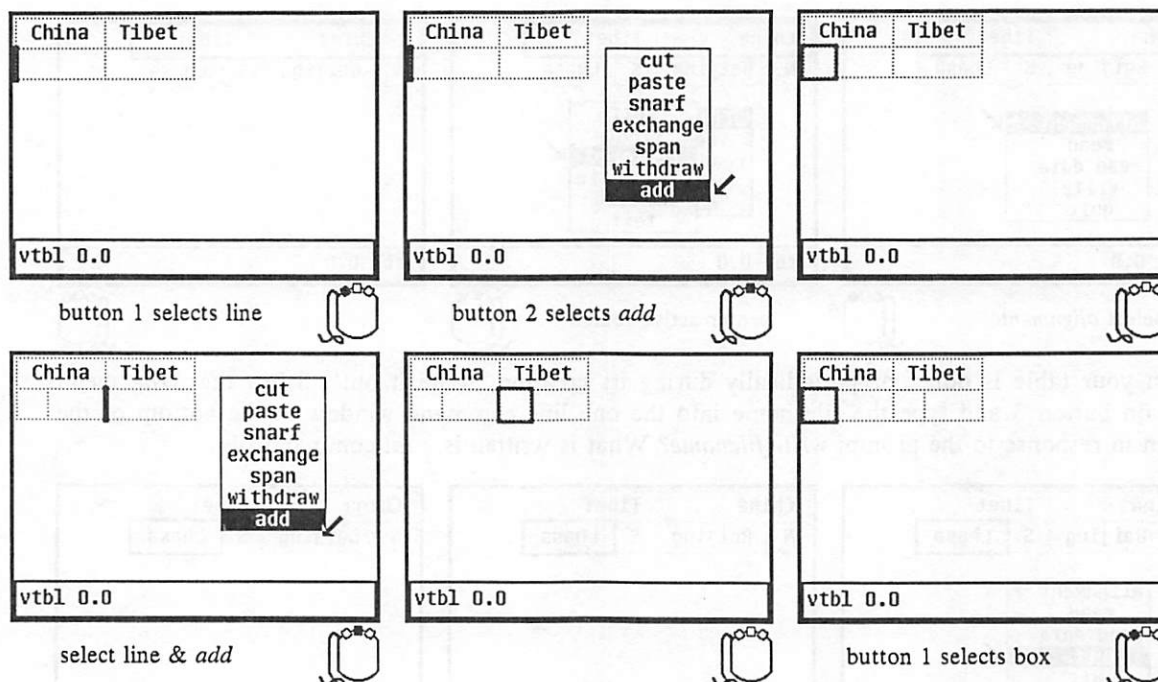
This time, rather than cutting and pasting paper, you want to try to get it into the computer. That's a noble ambition in this day and age. And if you have a DMD 5620 terminal, *vtbl* can help.

To design a new table from scratch invoke *vtbl* without arguments and it will provide you with a single active box on the screen. Typing a *tab* character will create a new column; typing a *return* will create a new row. Any other text will be placed in the box as data.



These three bitmaps were taken from the screen to illustrate table development. Throughout this paper, screen images supplement the text. Some have captions and mice which specify what the user is doing. The three little squares atop each mouse represent buttons 1 to 3 from left to right. A shaded button represents a pressed button. The left bitmap above shows the initial state of the table when the program is downloaded. Selected or active regions/lines are always highlighted. The framework of the table is displayed as a lightly-dotted grid. This grid does not represent lines in the final *tbl* output, but rather is a convenience to facilitate visual editing.

Typing "China" followed by a *tab* produces the middle table. The *tab* creates a new column, allowing you to continue entering table contents. Typing "Tibet" followed by a *carriage return* produces the table on the right. When the *return* creates a new row, *vtbl* copies the frame of the previous row and leaves you in the first column of the new row. You could simply continue typing data, *tab* and *carriage return* to finish a simple table. But in this case you want subcolumns. To add them, you need to specify where and what.



Use the mouse to select a line segment where a column is needed. Point to one end of the desired line segment and press button 1, holding it while moving along the line. Do not release the button until the segment you want, and only that segment, is highlighted. Select the button 2 menu item *add* and you've got your new subcolumn. This requires that you press button 2 to bring up the menu, drag the mouse until the item *add* is seen in reverse video and then release the button. This description may seem wordy in its attempt to be precise, but the task is simple. Try it.

You can intersperse typing table data and adding your columns as convenient; there are no order effects. Once you have added some columns, you can type into the active (most recently added) column or you can render another box active using the mouse. To select a box, point at it and click button 1. If you type when nothing is selected, nothing will happen. If something other than a single box is selected, *vtbl* will highlight the nearest box and add the text to that box. In the case of an active box group, it will highlight the upper left box of the group. In the case of a point or line, it will try the box to the right and below.

Here let us center the two place headings in our table. You can select both together and center them in one sequence. First use button 1 on your mouse to select the two boxes. Point to one box and press button 1 to select that box. Hold the button and sweep over the second box. Release the button when the pair of boxes are highlighted, as on the left below. Press button 3 to bring up a menu, hold the button while moving the mouse to the item *alignment*. Once alignment is seen in reverse video, sweeping to the right (as per the arrow) brings up the secondary menu of alignment types. Continue holding button 3 until the item *centered* is selected. When the button is released, the highlighted boxes will be realigned so as to center the text in each box.

China		Tibet	
N	Beijing	S	Lhasa

alignment →
read
read data
write
quit

vtbl 0.0

select alignment

China		Tibet	
N	Beijing	S	Lhasa

align
left
right
centered
read
read data
write
quit
alphabetic
numeric
text

vtbl 0.0

center active region

China		Tibet	
N	Beijing	S	Lhasa

vtbl 0.0

When your table is done, or periodically during its creation, write it out! Select the *write* menu item on button 3 and type the file name into the one line command window at the bottom of the screen in response to the prompt *write filename?* What is written is a *tbl* command file.

China		Tibet	
N	Beijing	S	Lhasa

alignment →
read
read data
write
quit

vtbl 0.0

write *tbl*

China		Tibet	
N	Beijing	S	Lhasa

write filename? china

type filename 'china'

China		Tibet	
N	Beijing	S	Lhasa

wrote china

1.2 Redesigning a table from scratch

Let's also look at a different way to do this 4-column table. Use the command *vtbl 2 4* to invoke *vtbl* with a 2 by 4 uninitialized table. Now what you need to do is span the headings in the first row and enter the data in the body.

--	--	--	--

vtbl 0.0

invoke *vtbl 2 4*

China			
-------	--	--	--

vtbl 0.0

type 'China'

China			
-------	--	--	--

cut
paste
snarf
exchange
span
withdraw
add

vtbl 0.0

button 2 selects *span*

China			
-------	--	--	--

vtbl 0.0

spanned columns 1 & 2

China	Tibet		
-------	-------	--	--

cut
paste
snarf
exchange
span
withdraw
add

vtbl 0.0

type 'Tibet' and span it

China	Tibet		
-------	-------	--	--

vtbl 0.0

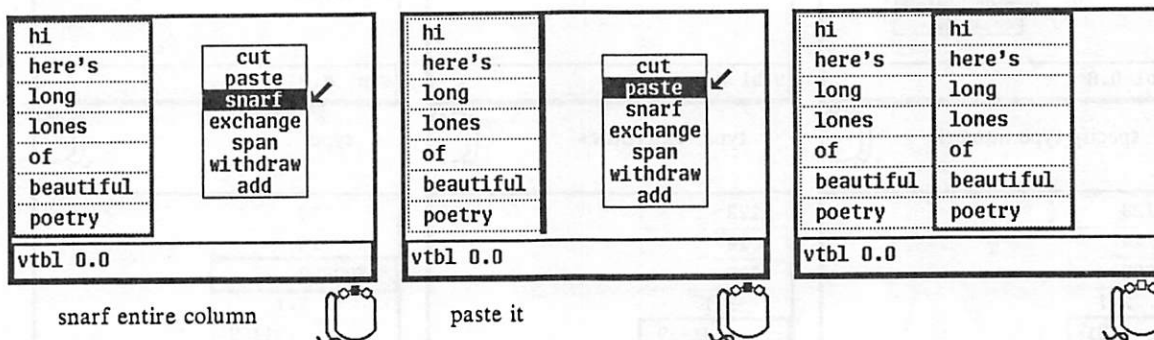
spanned columns 3 & 4

Once again, you begin with the first column in the first row highlighted. You can either type the heading into this box and then span it across the next two boxes, or you can span first and enter the text later. Let's type first, so the span operation will be easier to observe. To span, select the box which should span across others and then select the button 2 item *span*. The cursor is transformed into a little box with an arrow in the corner, which you can't see in the figures above. Pressing button 1 allows you to drag around the highlighted area and select the appropriate one. So, hold button 1 and stretch the highlighted area right across the next column. When you release the button, watch the columns quickly and correctly resize. The text from the box initially selected is repositioned in the new enlarged box. Repeat by selecting the next column and spanning it across the last column.

Vtbl continuously sizes columns as text is typed, pasted, spanned or cut. Similarly, it maintains correct alignment in alphanumeric and numeric columns as contents change. Additional width introduced by a spanning column is added to the last column spanned; this is a rule adopted from *tbl*. If the text in the spanning column is deleted, then the columns shrink as appropriate. In this table, cutting the text "China" would cause column 2 to shrink. Repasting would cause it to grow again.

1.3 Snarfing, cutting and pasting

Any box or set of boxes can be selected using button 1 and snarfed, cut or pasted using button 2. Snarfing simply copies the highlighted area into a single buffer without affecting what is in the table or on the screen. Cutting, on the other hand, cuts the highlighted area from the table displayed and squirrels a copy away in the buffer. Pasting takes whatever is in the buffer and inserts it into the table at the selected position. Boxes can be pasted over lines, which is a simple insertion. And boxes can be pasted over other box groups, in which case the highlighted area is first deleted from the table. Here's a simple case in which a single column is snarfed and pasted. Note that the newly pasted region remains highlighted so that a click of the button 2 *cut* item would remove it.



1.4 More on aligning and sizing

Data can be centered, right or left-justified, aligned as numbers, alphanumerics or text as defined by *c*, *r*, *l*, *n* and *t* formats in *tbl*. You can type the data and specify its alignment as convenient: typing all the data, and then sweeping regions to specify alignment or specifying that a specific box or region is numeric before typing the data. In either case, as new rows are created types are carried down columns.

Returning to the *poetry* example above, we can easily align the new right column so that it is right-justified.

hi	hi
here's	here's
long	long
lones	lones
of	of
beautiful	beautiful
poetry	poetry
vtbl 0.0	

align column

hi	hi
here's	here's
long	long
lones	lones
of	of
beautiful	beautiful
poetry	poetry
vtbl 0.0	

specify right-justification

hi	hi
here's	here's
long	long
lones	lones
of	of
beautiful	beautiful
poetry	poetry
vtbl 0.0	

1.4.1 Numerical alignment. Let's look at numerical alignment: at how numbers are aligned and consequently how column width changes as numbers grow. At the left below, you see numbers in a left-justified column. Once this column is declared to be type *numeric* using the mouse menu on button 3, the numbers are aligned at the decimal point or the units digit. If no alignment is indicated by the contents of a numeric box, then that item would be centered.

As a new number is typed, the column will need to grow depending upon the digits to the right and left of the decimal place independently, rather than the total width of the number. This set of bit-maps illustrates column growth as the user types. In maintaining proper alignment, adding numbers to the right of the decimal place causes the column to grow while adding to the left causes the column to grow and the numbers to shift right.

123
14
1500
3.1

align

left
right
centered
alphabetic
numeric
text

vtbl 0.0

specify type numeric

123
14
1500
3.1

vtbl 0.0

type '<RETURN>'

123
14
1500
3.1
.

vtbl 0.0

type '.'

123	
14	
1500	
3.1	
.31	
vtbl 0.0	

123
14
1500
3.1
.31459

vtbl 0.0

123	
14	
1500000	
3.1	
.31459	
vtbl 0.0	

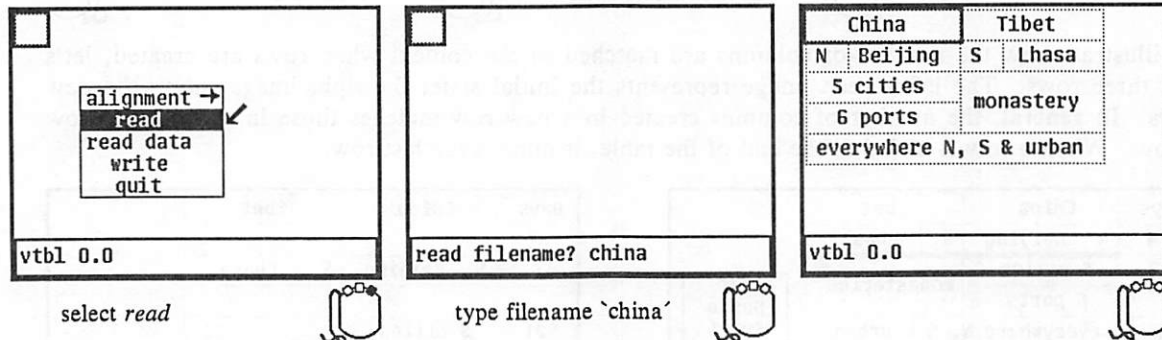
1.4.2 Other alignment types. Left-justified items always look the same; the columns may grow and shrink, but the text doesn't shift. All the other types are more inspiring to watch, because individual items shift around within the column depending upon its width. As a column grows, right-justified items shift right with each new character typed and centered items search for new centers. *Vtbl* continually updates column width and item position when each new character is typed and when regions are *cut*, *pasted*, *spanned* or *withdrawn*.

Quirks of *tbl* are respected wherever we have found them: the text "123...89" is aligned at the last decimal, while "..." is centered in a numerical column. Constraints of *tbl* are seriously enforced: error messages are given when a user tries to introduce a box of type *a* to a column containing

boxes of type *n*, and an assortment of messages are reported on improper *tbl* file input.

2. Revising a table

Now you've received the comments on your paper and need to make revisions, including those awful tables. Invoke *vtbl* with the table filename as an argument and it will read the *tbl* input and construct the corresponding table. Alternatively, you can invoke *vtbl* and then use the button 3 *read* item.



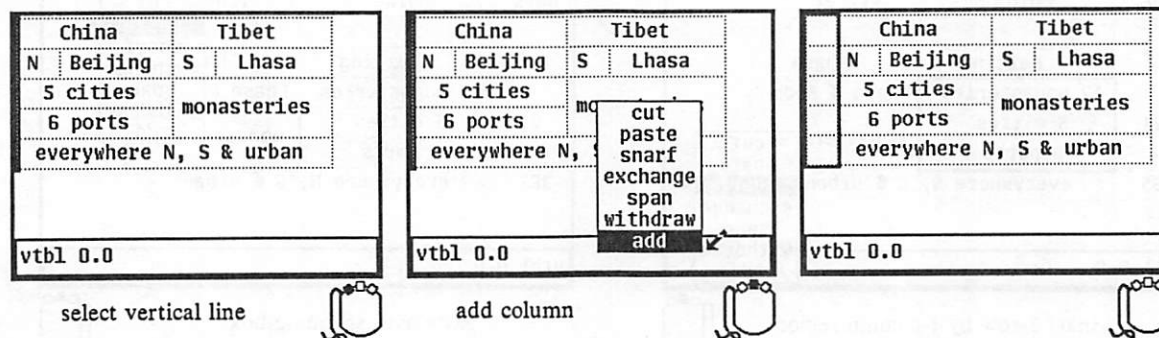
Now you can begin editing the existing table. You can cut a column, partially or fully withdraw a spanning box, change headings and correct data, add columns or subcolumns, cut and paste an entire subtable, or exchange two subtables.

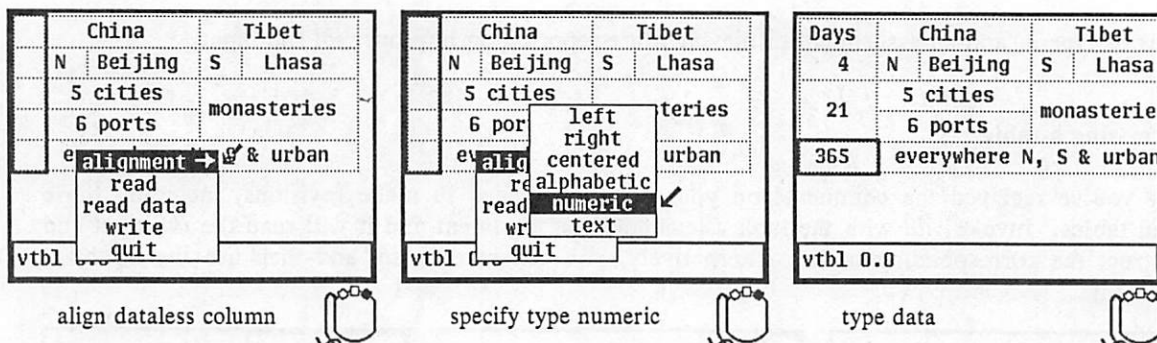
2.1 Entering and altering text

Currently the text entering is quite primitive, though our plans for the future are of course dramatic. *Tab* moves to the next column, *return* to the beginning of the next row. *Line feed* moves to the next item in the same column. Any printable characters typed appear in the active box. There are no editing capabilities beyond erase and kill, which delete the last character and the entire item respectively.

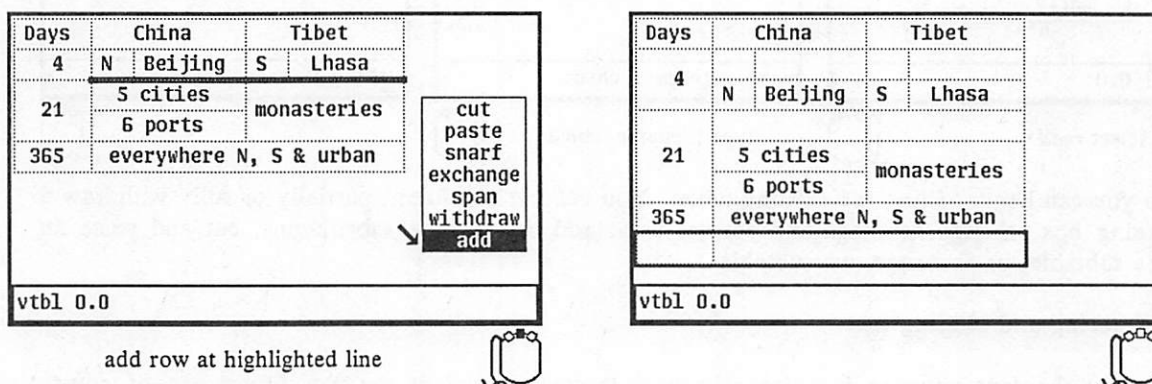
2.2 Adding columns and rows

These trips really do look marvelous, but nowhere is there information about the duration of the visits. Let's add a column at the left to specify the number of days. Note that when the table is larger than the window specified by the user, the table simply goes off screen. It cannot be operated upon, but it remains healthy. If something is cut or if the window is enlarged, it will reappear.





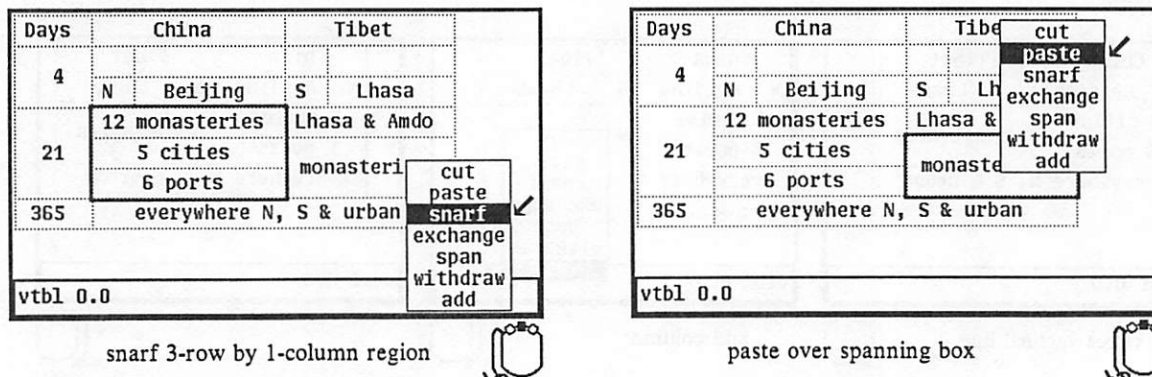
To illustrate how the number of columns are matched to the context when rows are created, let's add three rows. The left screen image represents the initial state; the right image shows the new rows. In general, the number of columns created in a new row matches those in the existing row below. When a row is added at the end of the table, it mimics the last row.



When columns are added, they also use the surrounding context to determine the number of rows.

2.3 Influence of context on pasting

Pasting a 3 by 2 subtable over one of the same size is straightforward. It is also common however, to paste regions of one size over regions of a different size. Vtbl is able to do this, and tries to match and stretch the surrounding region sensibly as shown in the six screen images below. In the first three bitmaps, a region which is 3 rows high is snarfed and pasted over a single spanning box. Three new boxes are thus created, and the box in last row to the left, which contains the text '6 ports', is stretched. In the last three, a two row region is snarfed and pasted over a 4 row region. This gets rid of some boxes and restores the earlier height of the stretched box.



Days	China		Tibet	
4	N	Beijing	S	Lhasa
21	12 monasteries		Lhasa & Amdo	
	5 cities		12 monasteries	
	6 ports		5 cities	
365	everywhere		N, S & urban	

vtbl 0.0

Days	China		Tibet	
4	N	Beijing	S	Lhasa
cut paste snarf exchange span withdraw add	monasteries		Lhasa & Amdo	
	5 cities		12 monasteries	
	6 ports		5 cities	
			6 ports	
	everywhere N, S & urban			

vtbl 0.0

snarf 2-row by 1-column region

Days	cut paste snarf exchange span withdraw add	China Beijing monasteries cities 6 ports everywhere	Tibet S Lhasa Lhasa & Amdo 12 monasteries 5 cities 6 ports N, S & urban
vtbl 0.0			

paste over 4-row by 1-column region

Days	China		Tibet	
4	N	Beijing	S	Lhasa
21	12 monasteries		Lhasa & Amdo 12 monasteries	
	5 cities			
	6 ports			
365	everywhere N, S & urban			

vtbl 0.0

Similar efforts are performed by *vtbl* for horizontal mismatches, as seen below. This illustrates a 1-row by 4-column snarfed region being pasted over a single simple box. Note that when that box is subdivided, it lines up with the two existing columns above. The last bitmap shows the newly pasted, and still active, region being cut. Note that cutting restores a frame derived from the surrounding context, so that the newly introduced boxes are gone.

Days	China		Tibet	
4	N	Beijing	S	Lhasa
21	12 monasteries		Lhasa & Amdo	
	5 cities		12 monast	
	6 ports		cut paste snarf	
365	everywhere		N, S & urb	exchange span withdraw add

vtbl 0.0

snarf 1 by 4 region

Days	China		Tibet	
4	N	Beijing	S	Lhasa
21	12 monasteries		Lhasa & Amdo	
	5 cities		12 monasteries	
	6 ports		5 cities	
365	everywhere N, S & urban			

cut

paste

snarf

span

exchange

withdraw

add

vtbl 0.0

paste over single box

Days	China				Tibet	
4	N		Beijing		S	Lhasa
21	N		Beijing		S	Lhasa
	5 cities				Lhasa & Amdo	
	6 ports				sterie	
365	everywhere N, S				cut paste snarf exchange span withdraw add	
vtbl 0.0						

cut new region

Days	China		Tibet	
4	N	Beijing	S	Lhasa
21			Lhasa & Amdo	
	5 cities			
	6 ports		12 monasteries	
365	everywhere		N, S & urban	

vtbl 0.0

Note that the entire table can be cut, in which case, the display degenerates to a 1 by 1 table with no data. The entire table can be snarfed. By then selecting the bottom horizontal or the rightmost vertical line and pointing to the *paste* operation, the entire table can be replicated at the bottom or to the right. This can be really convenient when subtables repeat to form a large table.

3. Future plans

We have recently heard of another visual table program, *Intertable*,⁶ which provides a non-bitmapped visual interface to *tbl*. Their paper describes careful width calculations, but we have not yet seen their user interface. We hope to be able to do so soon.

We have code in place for allowing the user to add horizontal and vertical lines, both within and across rows and columns; we should debug it so it doesn't crash the program... We are in the process of adding font selection, which will incorporate an existing and rather elegant font package as well as proper handling of long text items, a la $T\{ \dots \}T$. We will one day add some true editing capabilities. There have been requests for the ability to specify whether data files should be read by row or by column (now files are interpreted by row). Users have suggested that it would be useful to be able to read *tbl* commands or data files into a selected region and, analogously, to be able to write only the selected region. Finally, we need to add an *undo* command, most importantly to escape death at the hands of certain careless users.

REFERENCES

- [1] Kernighan, B. W., "PIC — A Language for Typsetting Graphics," *Software Practice & Experience* 12(1), pp. 1-21 (1982).
- [2] Bentley, J. L. and Kernighan, B. W., "GRAP — A Language for Typesetting Graphs," *Computing Practices* 29(8), pp. 782-792 (1986).
- [3] Browning, S. A., "Cip User's Manual: One Picture is Worth a Thousand Words," AT&T Bell Laboratories internal memorandum (March 19, 1982).
- [4] Pike, R., "The Blit: A Multiplexed Graphics Terminal," *Bell Labs Technical Journal* 63(8 Pt. 2), pp. 1607-1631 (1984).
- [5] Lesk, M. E., "Tbl — A Program to Format Tables," UNIX Programmer's Manual 2, Section 10 (January 1979).
- [6] Van Vliet, J. and Warmer, J., "Intertable," Report CS-R8636, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands (November 1986).

Psfig — A DITROFF Preprocessor for POSTSCRIPT Figures

Ned Batchelder†

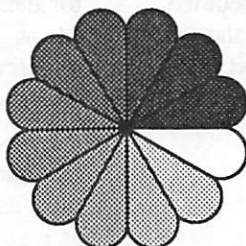
Trevor Darrell



Computer and Information Science Department
University of Pennsylvania
200 South 33rd Street
Philadelphia, PA 19104

ABSTRACT

Psfig is a new preprocessor for TROFF. It implements a general figure inclusion, where a figure is any POSTSCRIPT file. For example:



Figures are automatically scaled and positioned, with all sizes under user control. *Psfig* can be used not only for actual figures, but also to provide special effects in standard text, like **white on black** or custom special characters, like 'ø'. For speed and for compatibility with non-POSTSCRIPT systems, a draft mode is available that simply shows the name of the POSTSCRIPT file and the extent of the figure.

1. Introduction

The POSTSCRIPT language is a powerful page description tool that is rapidly becoming a *de facto* standard, and is available in printers with a wide range of price and performance from several manufacturers. Since TROFF has a long history of preprocessors that provide new functions (*eqn*, *tbl*, *pic*, and *grap* provide equations, tables, line drawings, and graphs respectively), we wanted to write a preprocessor that would provide the capa-

bility to include arbitrary POSTSCRIPT figures into a document. *Psfig* is the result.

1.1. Simple Use

The simplest *psfig* command is simply the word *figure* followed by the name of a file. If we have a file called 'rosette.ps' which contains the POSTSCRIPT code to draw the rosette in the abstract, we would use the *psfig* command

```
figure rosette.ps
```

to include it as a figure. (We'll explain how to combine *psfig* commands with the rest of a TROFF document in a little while.) *Psfig* will automatically position the figure to the proper

† Author's current address: Digital Equipment Corporation, 129 Parker Street, PKO3-1/K90 Maynard, MA 01754

place on the page, regardless of its 'natural' position. It will also instruct TROFF to reserve the space occupied by the figure so that it doesn't overlap with anything else on the page.

Because no mention of size was made, *psfig* draws the figure at its natural size. The rosette's natural size is about 4 inches across, which is a little large; the rosette in the abstract was produced with:

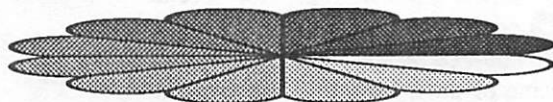
```
figure rosette.ps height 1.25i
```

The height clause specifies how high the figure should be. We've asked for it to be 1¼ inches high. The word 1.25i is interpreted by TROFF, so any expression that TROFF can evaluate is acceptable. For explicit measurements, the units i, c, and p for inches, centimeters and printer's points (1/72 inch) are available (among others) for absolute distances, and m, n, and v for the current point size, half the current point size, and the current line spacing are available for distances that vary according to their environment.

Since the width of the figure wasn't specified, it was scaled equally so that the shape of the figure is maintained. Of course, if desired, both dimensions can be explicitly specified. The command

```
figure rosette.ps
      height .5i
      width \n(.1u
```

produces:



(The width expression \n(.1u is a TROFF incantation that means the current line length. It stretches the rosette across the column exactly.)

Multiple figure commands place figures side-by-side across the page, and space between such figures is specified with a space command.

```
figure rosette.ps height .5i width .2i
space .5i
figure rosette.ps height .5i width .8i
```

produces:



Figure 1 is an overview of all of the components in the system, and by the way, a fairly complex example of what can be done with *psfig*.

2. Design Overview

In designing *psfig*, our first goal was to serve the average TROFF user who desires to include figures in a document with the power and generality of POSTSCRIPT. The existing TROFF preprocessors do a fine job in each of their specialized tasks, but none of them approach the generality or descriptive power of POSTSCRIPT. We saw *psfig* as being the base for a broad range of uses, so in addition to being easy to use, it had to be powerful. We wanted the capability to include anything from a full page high resolution image down to a special mathematical symbol, as well as also to provide an ability to pass through literal POSTSCRIPT for special effects.

2.1. Interface Design

Good user interfaces are as simple as possible in the base case, yet retain full generality for more sophisticated use. Our goal for *psfig* was to design a user interface that is extremely simple for the simplest case of including a POSTSCRIPT file as a figure, but that has the power necessary to describe more complex operations that sophisticated users might want.

We also set out to make sure that it was a familiar interface. We explicitly tried to make *psfig* as much like *eqn* as possible. We saw *eqn* as a good example of a powerful yet easy to use preprocessor which solved most of the problems of interface design that we were going to face.

Lastly, we wanted to make sure that *psfig* was as powerful as possible. In the spirit of the other TROFF translators, as much information as possible is simply passed through the preprocessor. For example, we chose not to define our own units for the specification of figure dimensions. Rather, we assume that the expression used will be interpretable by TROFF, and pass it through. This provides a powerful link between *psfig* and TROFF, and allows for uses that would have been impossible had *psfig* interpreted all dimensions itself.

2.2. Figure Placement

To most users, the operation of including a figure into a document seems a natural one, and the behavior that should result from simple figure inclusion commands seems obvious. We wanted to make sure that *psfig* would conform to these expectations that people had about figure placement.

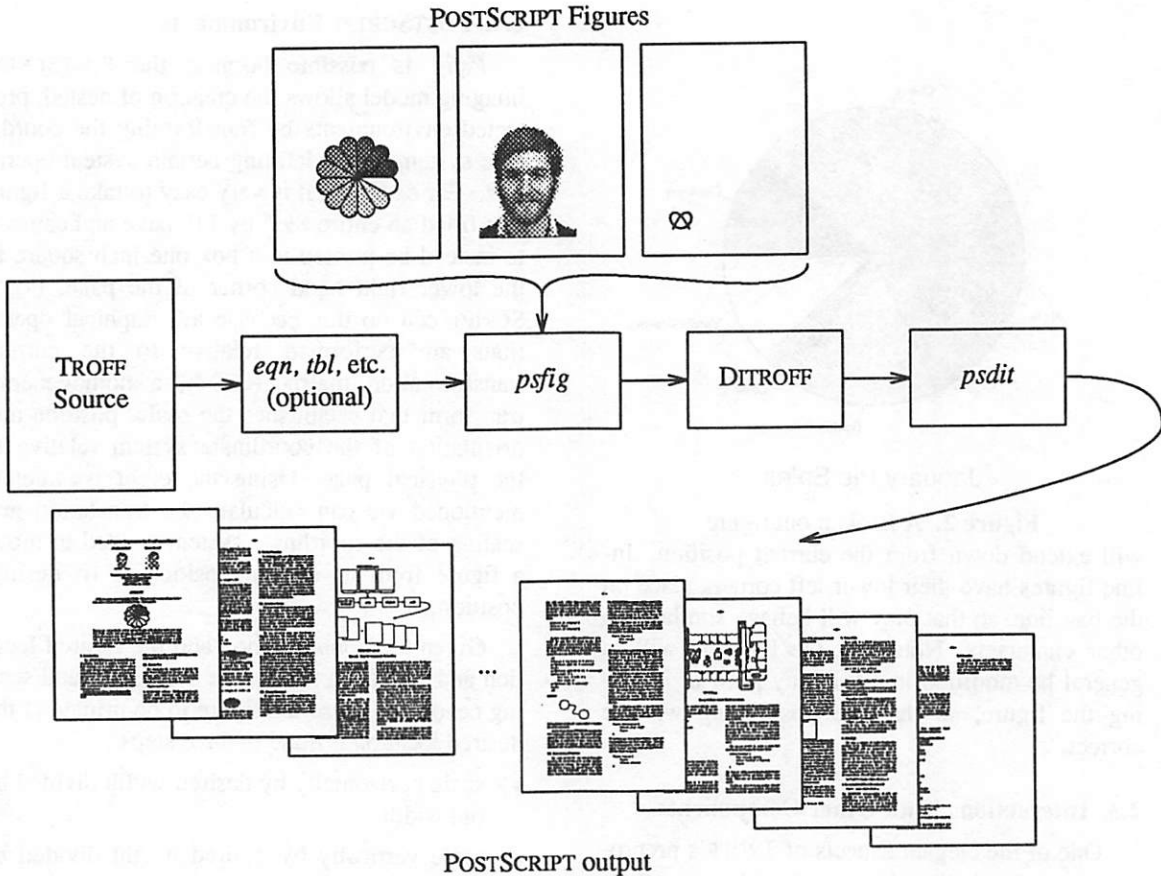


Figure 1. How the components fit together.

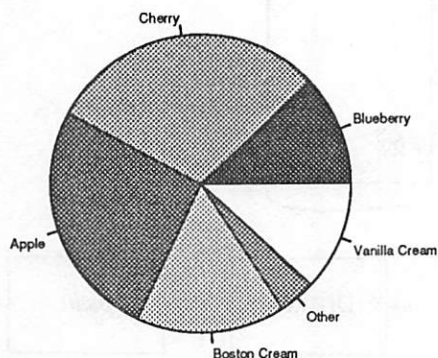
There are eight values that govern how *psfig* manipulates a figure to position it properly: four for where the figure lies in its natural POSTSCRIPT coordinate system, and four for where the figure is desired to fall on the page. We use the term “natural” position to refer to the size and location of the figure would have if it were printed alone, and we adopt Adobe’s convention of describing a figure’s natural size with a bounding box, specifying the x and y coordinates for both the lower left and upper right corners.

To avoid confronting this collection of numbers for every figure, we developed some defaults to use when positioning a figure. We assume the desired position of the figure is at the current TROFF pen position, and that the figure should keep its natural height and width. If the figure conforms to the POSTSCRIPT Document Structuring Convention as defined by Adobe Systems, the figure’s natural size (and position) will be declared in a POSTSCRIPT comment which *psfig* can read. More often than not, though, one does not want the figure at its natural size. We designed *psfig*’s figure manipulations to take into

account not just a translation to move the figure to the proper place on the page, but also a scaling to resize the figure. *Psfig* allows you to specify a height and/or width for the figure. If only one is given, the user’s expectation is that the other will be calculated to maintain the figure’s original aspect ratio, so this is what *psfig* does.

Because users will find varied purposes for *psfig*, it understands about two different types of figures. Broken out figures are similar to *eqn*’s display equations: they reserve space across an entire column so that they occupy space like paragraphs. They are used for actual figures in documents. The pie chart in Figure 2 is an example of a broken out figure. In-line figures are similar to *eqn*’s in-line equations: they occupy space within the current line of text, and are used to provide special characters for unusual applications. Examples include the pretzel (∞) from the abstract.

Because these two types of figure have different uses, they have different defaults for positioning. Broken out figures have their upper left corner placed on the TROFF baseline, so that they



January Pie Sales

Figure 2. A Broken out figure will extend down from the current position. In-line figures have their lower left corners place on the baseline, so that they will behave similarly to other characters. Note that the baseline will in general be modified in some way prior to invoking the figure, so that the positioning will be correct.

2.3. Interactions With Other Components

One of the elegant aspects of TROFF's preprocessor system is that there seem to be no restrictions on their use with each other. Any or all of them can operate on different or even the same parts of the document without any ill effects. We worked hard to maintain this standard of compatibility in creating *psfig*. It has been successfully used with the standard preprocessors *tbl*, *eqn*, and *pic*, just as TROFF users would expect (uses with *eqn* and *pic* are included in this paper).

Finally, we sought to make *psfig* interact well with other variants of TROFF. To be able to pass commands through to the output file, *psfig* requires full DITROFF, to which many people do not have access. It also obviously presupposes a POSTSCRIPT output device. In order that source files using *psfig* can be processed with other systems (for example, NROFF or vanilla TROFF without special postprocessors or POSTSCRIPT), a draft mode feature has been included which uses no unusual features of either the formatter or the output device. Of course, none of the POSTSCRIPT figures are included in the output, but indications of them are given, and the formatting of the TROFF output around them is the same.

2.4. POSTSCRIPT Environments

Psfig is possible because the POSTSCRIPT imaging model allows the creation of nested, protected environments by transforming the coordinate system and redefining certain system operators. For example, it is very easy to take a figure that filled an entire 8½" by 11" page and cause it to instead be printed in a box one inch square in the lower right hand corner of the page. POSTSCRIPT can do this because all graphical operations are performed relative to the current transformation matrix (CTM), a homogeneous transform that establishes the scale, position and orientation of the coordinate system relative to the physical page. Using the eight parameters mentioned we can calculate the translation and scaling of the coordinate system needed to move a figure from its natural position to its desired position.

Given the bounding box and the desired location and size for a figure, the translation and scaling needed to cause the figure to be printed at the desired location is done in three steps:

- scale horizontally by desired width divided by old width
- scale vertically by desired height divided by old height
- translate the upper left hand corner of the figure's bounding box to the current point.

POSTSCRIPT also gives us the tools necessary to insure that any side effects of a figure do not affect the rest of the document. The POSTSCRIPT operators *save* and *restore* effectively undo the side effects of any code executed between them. *Psfig* brackets all figures with these operators to protect the document. The environment in which the POSTSCRIPT code for a figure is executed places no restrictions on the commands which may be used, so any well formed (and non-hostile) POSTSCRIPT file can be included as a figure. The POSTSCRIPT operators *showpage*, *initgraphics*, *initmatrix*, and *defaultmatrix* are locally redefined for the figure so they behave in a rational way. For example *initgraphics* first performs a 'regular' *initgraphics*, but then restores the current transformation matrix (CTM) to the one we created for the figure. The redefinition of *showpage* is simply to do nothing. It is expected that no multi-page POSTSCRIPT files will be included as figures, so this redefinition really amounts to ignoring any *showpage* that may appear at the end of the figure.

3. The Preprocessor

The *psfig* preprocessor is responsible for the interface seen by the user. Essentially it translates a higher-level syntax into file inclusion and literal POSTSCRIPT calls which are passed through DITROFF and interpreted by the postprocessor. Like the other TROFF preprocessors, *psfig* interprets those portions of the file that are marked as its input. This input it translates into raw TROFF code. All other portions of the file are passed directly through to the output, to be interpreted further down the line.

The basic *psfig* command to include a figure is the optional keyword *figure*, followed by the name of a file containing a POSTSCRIPT program, followed by any number of optional clauses. Some common clauses are:

```
height h
width w
bounds llx lly urx ury
```

which specify the size of the desired figure, and the bounding box of the original figure. As promised, if there is no *bounds* clause for a figure, *psfig* scans the POSTSCRIPT file for the bounding box comment, and will also compute the height and width using the defaults discussed earlier.

Psfig also provides an lower level interface with the *file* and *literal* commands, which provide direct file and literal POSTSCRIPT inclusion, respectively. Finally, a *global* option is available on *file* and *literal* to download code that will remain present across the *save* and *restore* context normally surrounding each DITROFF page.

In the simplest usage then, one need only specify the name of a file containing POSTSCRIPT to include a figure, and *psfig* will perform a default set of "reasonable" actions.

The full input syntax is included in Appendix A.

4. DITROFF and Postprocessor Hooks

The preprocessor bears the brunt of making things look and act in a well behaved manner, but the real work is done in the postprocessor and its POSTSCRIPT prolog files.

Psfig uses the DITROFF *\X* command to pass commands through to the postprocessor. An input sequence of

```
\X'test'
```

will come through DITROFF as

```
x X test
```

We used *psdit*, the DITROFF to POSTSCRIPT translator supplied with TranScript from Adobe Systems as our postprocessor. We added two primitives that we call through *\X*:

```
\X'f filename'
\X'p literal POSTSCRIPT'
```

The former interpolates the contents of *filename* into the POSTSCRIPT output *psdit*, while the latter injects *literal POSTSCRIPT*. In passing literal arguments, we trick DITROFF into evaluating dimension expressions for us by enclosing our expression in *\w'\h'expr'*.[†] This will evaluate to the value of *expr* in device units. All scaling computation is done in this way, since the preprocessor can not know the value of TROFF variables, which may well be used in expressions. This also means the preprocessor need not know anything about TROFF dimensions, and users can specify dimensions in the same ways they always have. Inside *psfig*, the computation is manipulated in an algebraic manner, and is finally evaluated when it passes through *\X*.

5. Putting it all Together

Now we can examine exactly how we create the nested and protected environment. We perform a simple figure inclusion in three steps:

- Using *psdit*'s new literal pass through command, construct a call to a 'startFig' POSTSCRIPT function that we have included in the prolog prepended to all POSTSCRIPT files from *psdit*. The *startFig* function takes the desired height and width, and natural bounding box as arguments, issues a *save*, performs the necessary transformations of the graphics state, then redefines system operators as needed.
- Using *psdit*'s file inclusion command, copy the figure file into the output stream.
- Using *literal*, call our 'endFig' POSTSCRIPT function that undoes the effects of 'startFig'. *EndFig* needs no arguments.

So, to show a small example, if we had a POSTSCRIPT figure in a file *smiley.ps* that contained the code:

[†] This TROFF incantation asks for the overall width of a string that is nothing but a relative horizontal motion by *expr*.

```
%!
%%BoundingBox: 0 0 36 36
newpath 18 18 10 0 360 arc stroke % head
newpath 18 18 6 180 360 arc stroke% mouth
newpath 22 22 .5 0 360 arc stroke % eyes
newpath 14 22 .5 0 360 arc stroke
showpage
```

and *psfig* was processing the TROFF source fragment:

```
for a happy document!
.F+
figure smiley.ps
.F-
.NH 1
What is a Figure?
```

psfig would translate 'figure smiley.ps' into a series of \X calls which would cause the following POSTSCRIPT output from *psdit*:

```
760 4512(happy)N 976(document!)X
1422 4560 MX
288 288 0.00 0.00 36.00 36.00 startFig
...contents of smiley.ps...
endFig
3 f 760 5040(6.)N
860(What)X 1071(Is)X 1153(A)X
1231(Figure?)X
```

The height and width are the first arguments to *startFig*, and are in DITROFF device units, followed by the natural bounding box of the figure in points. *startFig* will convert the height and width into points, then perform the computation outlined above. And all this makes for a happy document!



6. What Is A Figure?

Since figures are simply POSTSCRIPT files, *psfig* allows dozens of utilities to be figure tools. Most graphical tools either directly produce POSTSCRIPT, or produce an output language (such as Tektronix 4014, and Unix plot) that can be translated into POSTSCRIPT using available filters.

6.1. Figure Requirements

Of course, in addition to the POSTSCRIPT information about the appearance of the figure, *psfig* will need some information about the figure that it can use for computing the size of the fig-

ure. The only requirement on a figure is that it produce valid POSTSCRIPT code, and that it contain a %%BoundingBox comment as described in Adobe's Document Structuring Conventions. *Psfig* insures that a figure is actually POSTSCRIPT by checking that the first two characters in the file are '%!'.

Note that the mere presence of the proper identifying characters and a bounding box comment will not insure a figure will behave properly. There are many ways a POSTSCRIPT program could fail in a *psfig* environment, from having an erroneous bounding box, to causing a POSTSCRIPT error when executed, to circumventing *psfig*'s redefinition of system operators, to using operators that haven't been protected by *psfig*, such as *exitserver*. As with most trap door mechanisms that allow arbitrary information to pass through a processor, *psfig* has little choice but to trust the figures it deals with. It makes some minimal checks that catch the most blatant problems, but it cannot do more than that.

6.2. Encapsulated POSTSCRIPT

Recently Adobe has addressed the issue of POSTSCRIPT programs that are designed explicitly for use as included figures, and have established an *Encapsulated POSTSCRIPT File Format*. Part of this standard deals with file formats for dual bitmap/POSTSCRIPT representations, primarily for Macintosh and MS-DOS applications, and is not important for this discussion. Other parts, however, present guidelines for safe POSTSCRIPT code that can be imported into documents, and they outline some of the techniques that are used by *psfig* to set up a nested environment. The standard does not assume any operator redefinition other than *showpage*, and it provides a list of operators that seriously disturb the state of the interpreter, and are forbidden in conforming EPSF programs:

Operators to avoid in imported files (EPSF 1.3)	
exitserver	initgraphics
initmatrix	initclip
erasepage	copypage
grestoreall	framedevice
setpageparams	banddevice
nulldevice	renderbands
note	

In particular, only operators documented in the

body (that is, not an appendix) of *POSTSCRIPT Language Reference Manual* (commonly known as the Red Book) should be used since the availability of any others cannot be guaranteed in all POSTSCRIPT implementations. In general, any POSTSCRIPT file used with *psfig* should conform as closely as possible to the EPSF specification. Note that even though *initgraphics* and *initmatrix* are redefined for *psfig* figures, their use is discouraged.

7. Tutorial

7.1. TROFF Interface

Like the other TROFF preprocessors, *psfig* passes most of its input through to its output untouched. Only text that is marked as a *psfig* command is interpreted.

There are a number of ways to mark *psfig* commands in your TROFF document. The first is to enclose them between *.F+* and *.F-*:

```
.F+
psfig commands
.F-
```

This is precisely equivalent to *eqn*'s *.EQ* and *.EN*: The *.F+* and *.F-* lines are copied through to the output so that macro packages can do some action before or after figures. Any arguments to *.F+* or *.F-* are copied through to the output but are otherwise ignored. In our definitions of these macros, *.F+* and *.F-* provide a displayed figure centered in the line, and giving *.F+* an argument of *L* will leave the figure left flush.

Like *eqn*, *psfig* has the ability to read commands from within a TROFF line. The *delim* command specifies two characters that will delimit *psfig* commands:

```
.F+
delim @@
.F-
```

Any text that falls between the two characters specified will be interpreted as commands by *psfig*. In-line commands are most useful for generating special characters like the pretzel in the abstract, because they don't cause a break in the text. One restriction: an in-line command must not be broken across two lines.

7.2. Command Structure

Psfig commands consist of words separated by white space (spaces, tabs, or newlines). Some words, like *figure* and *space* are reserved words, and mean something to *psfig*, while others, like *rosette.ps* and *\n(.lu* are assumed to mean something to someone else. (In this case, the file system and TROFF).

A command that starts with a non-reserved word is assumed to be a *figure* command, so the word *figure* can usually be omitted. Semicolons are taken as command separators and can be used to avoid ambiguities caused by the omission of a reserved word.

Because non-reserved words aren't interpreted, they must be quoted if they contain any characters that *psfig* interprets specially. Either single or double quotes may be used. One exception: it is impossible for an in-line command to contain the closing delimiter character, even if it is quoted.

7.3. In-line Figures

Figures that result from in-line commands are slightly different from figures created the other two ways. First, whereas broken out figures have their top edge on the current baseline and extend down, in-line figures sit with their lower edge on the current baseline, and extend up. This facilitates the use of in-line figures to create custom characters like the pretzel (\wp). For example, the last sentence ended with:

```
pretzel (@ pretzel.ps width 1.3n @) .
```

The width here is specified in the TROFF unit *n*, which is the width of a lower case 'n' in the current point size. Specifying a width this way makes the character the right size regardless of the current text size:

A larger pretzel: ' \wp '.

Characters designed this way can be used anywhere a standard character can be used:

$$\sum_{i=0}^{\infty} x^{\wp} = 2 \sin(\wp)$$

One hint for use with *eqn*: always enclose *psfig* commands with quotes when inside *eqn* commands. For example, part of the above equation was created with

```
x sup "@ pretzel width 1.3n @"
```


Another difference between in-line and broken out figures is that by default, in-line figures don't reserve any vertical space, under the assumption that they will fit within the current line anyway. If your in-line figure is higher than anything else on the line, and you want the space to be reserved, then add the word `reserve` to your command. If your figure isn't higher and you use `reserve`, the spacing will be wrong, so only use it if you need it.

7.4. Macros

Psfig provides a macro facility that is similar to *eqn*'s. A command of the form:

```
define foo /bar/
```

will define a macro named `foo`. Any occurrence of the word `foo` will now be replaced by the word `bar`. The text of the macro is delimited by any character not included in the text itself, and may be any sequence of characters, including any of the characters that *psfig* interprets specially.

Macros can be useful for commonly used figures like in-line characters. For example, this manual begins with the following lines:

```
.F+
delim @@
define wd /width/
define pretzel /pretzel.ps wd 1.3n/
.F-
```

and all the pretzels in the text were created with:

```
@pretzel@
```

Remember that using a width specified in `n`'s gives us size independence, so that this macro will work in any environment to give us the right size pretzel.

Macro expansion is attempted for every word that *psfig* sees, unless it is quoted. In particular, the name of a macro in a `define` command is expanded if possible, so be careful about redefinitions. The best policy is to always enclose the name in quotes:

```
define "wd" /width/
```

Macros in the text of a macro are expanded when the macro is expanded, not when it is defined.

7.5. Special Effects

Psfig can also be used to provide interesting graphical effects.

For example, this paragraph has been printed on a gray background. We diverted the text of the paragraph, scaled a gray box to fit around it, and then printed the text on top of the gray.

The command used to create the gray box on which the text sits was:

```
.F+ L
figure gray.ps
height \n(dnu+ln
width \n(.lu
reserve 0 0
.F-
```

The file 'gray.ps' draws a unit square filled with a light gray. We specified the height to be a little bit more than the height of the last diversion (the paragraph), and the width to be the same as the width of a line.

The `reserve` clause tells TROFF how much space to reserve, here, none. Normally, *psfig* has TROFF reserve the space taken by the figure so that it won't overlap with anything else. Here we want it to overlap, so we override the default, and have TROFF reserve no space. The `L` on the line with `.F+` overrides the default centering, so that the box is flush left.

7.6. Raw POSTSCRIPT

In addition to specifying files to include into the DITROFF output, you can also specify literal POSTSCRIPT text to be output. The basic command is

```
literal /text to be output/
```

The text (which is delimited by any character, just like the text of `define` commands) will be inserted into the POSTSCRIPT output without any protection around it. No macros are expanded in the literal text, but interpretation of TROFF constructs is performed.

Because there is no protection, you must be careful when writing literals. Any modifications you make to the state of the POSTSCRIPT interpreter will linger into the rest of your document. Also keep in mind that the POSTSCRIPT text is interpreted in the environment of your document, not a special figure environment, so any output generated will probably be wrong.

Because of this, literal text is really designed to be used as a way to output small amounts of POSTSCRIPT code to modify the way something

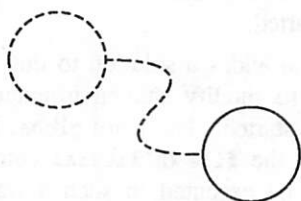
else will work, rather than generating output itself. For example, the white on black effect in the abstract was produced in part by bracketing the words 'white on black' with some literals that change the color to white and then back to black:

```
@ literal /1 setgray/ @
white on black
@ literal /0 setgray/ @
```

Another example of the use of `literal` is to modify the way lines are drawn by `pic`. Normally, `pic` allows simple dashed or dotted lines, but not complex dash patterns or dashed splines. By using literals to change POSTSCRIPT's dash parameter, you can achieve these effects:

```
.PS
define ps | box invis ht 0 wid 0 |
ps "@literal /[25 15] 0 setdash/@"
circle
ps "@literal /[20 15 40 15] 0 setdash/@"
spline right .5 \
    then down .5 left .5 \
    then right .5
ps "@literal /[] 0 setdash/@"
circle
.PE
```

produces



Notice that we used a `pic` macro called `ps` to hide the `psfig` commands.

Another form of raw POSTSCRIPT output is the `file` command, which takes the named file and outputs it at the current point with no protection. No scaling or positioning is done, so in general, the file should not produce any output, since it will not be able to predict its position on the page.

7.7. Preludes And Postludes

The main use for the raw output forms discussed in the last section is to provide auxiliary information for a figure. For example, let's suppose that you have a file named 'fig.mac' which contains some POSTSCRIPT output from MacDraw. Since Macintosh applications assume that

the POSTSCRIPT they generate will be preceded by a header file ('mac.pro') full of function definitions that the application can make use of, fig.mac will not work properly without the header. One solution would simply be to modify fig.mac by copying the header file into the beginning of it.

Rather than force you to do that, `psfig` provides you with a way to specify the relationship between fig.mac and mac.pro. Our current example would be specified like this:

```
figure fig.mac {
    file mac.pro
    figure
}
```

The braces enclose a list of things to be output in the order they should appear. We name the file 'mac.pro' first, so it is output first. Then the word `figure` by itself means the figure named at the beginning of the command. Both of these are enclosed in one environment. An example of MacDraw output is at the top of the next page.

The list of things to output can be placed anywhere in the `figure` command, even before the file name of the figure, and may contain any number of entries, although the figure must be referred back to (by the word `figure`) at most once. The entries (aside from the `figure`) can be either files or literals, and may appear either before or after the `figure`. If the word `figure` doesn't appear it is assumed to be the last item in the list.

This mechanism provides a general way to modify the behavior of figures. For example, a figure could be designed so that it reads arguments off the POSTSCRIPT stack, with a `literal` providing them at run time:

```
figure takesargs.ps {
    literal /arg1 arg2/
    figure
}
```

Or perhaps you have a shape that you want outlined sometimes and filled sometimes. You can put the commands to create the path into a file called 'logo.ps' and then make use of a `literal` after the figure to draw it:

```
figure logo.ps {
    figure
    literal /stroke/
}
```

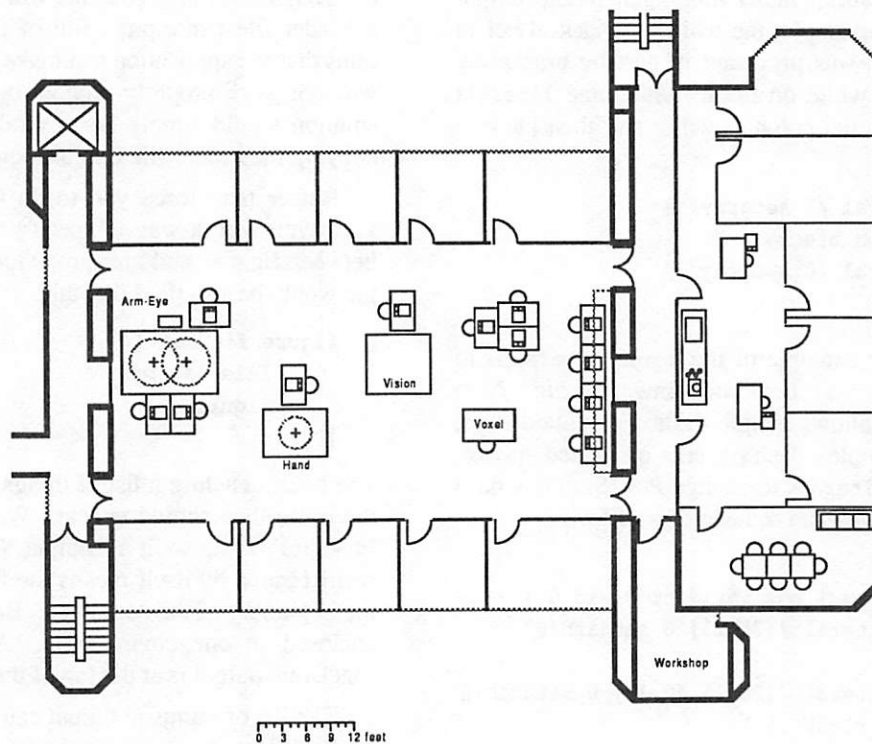



Figure 2. Some sample MacDraw output.

This feature can be coupled with the macro definition feature in a clever way. If you are going to be dealing with many MacDraw figures, you could define a macro:

```
define "macfig" /
figure {
    file mac.pro
    figure
}
/
```

and then simply say

```
macfig fig.mac
```

to include the figure.

7.8. Global Data

The above technique for including MacDraw documents points up a problem: the header file will be downloaded for each figure that needs it. Since the header file can be quite large (mac.pro is more than 25K bytes), this could get quite wasteful.

One solution would be to download the header once, and then to just download each figure separately. This will work except that each DITROFF page is an isolated environment, and

each page begins with the environment that the entire document began in. The header file will be available to every figure on the page in which it was downloaded, but will be lost when another page is started.

Psfig provides a solution to this by allowing the user to modify the environment in which pages are started. The word `global` can be used to modify the file or literal commands, and they will be executed in such a way that their effects are seen throughout the rest of the document.

We can use `global` to create a macro that does the work of loading the header for us:

```
define "macfig" |
    file mac.pro global;
    define "macfig" / figure /;
    figure
|
```

The first use of the macro downloads the header file, redefines `macfig`, and begins a figure command. Other uses are then simply `figure` commands.

Careful use of `globals` can produce interesting results, but care must be taken. For example, because successive pages depend on `globals` on


previous pages, the pages of the document cannot be reversed and still print properly.

7.9. Clipping

Normally, no clipping is done on figures; they are trusted to print only within their declared bounding box. If clipping is desired, the word `clip` can be added to a `figure` command, and the figure will be clipped to its bounding box.

7.10. Draft Levels

Because some POSTSCRIPT figures can be expensive to print (half-toned pictures, for example), and because documents designed to be printed on POSTSCRIPT printers may have to be printed on less capable printers, *psfig* allows the user to control the extent of the inclusion of figures. Every figure has associated with it a 'level', which should correspond roughly to the cost of printing it. When *psfig* processes a file, it runs at a certain level, and figures whose cost is less than the current level get printed. Broken out figures whose cost is more than the current level are omitted, and a box is drawn around where they would be:



The box has the name of the file in it for identification. In-line figures are simply omitted, but the space they occupy is still reserved by TROFF. Here is a draft pretzel: ' '. Since a box takes less time to draw than a complicated figure, the careful use of draft levels can speed up the printing of your document. Also, the box is drawn with standard TROFF commands, so by setting *psfig* to run at the lowest level (so that it decides that all the figures are too expensive), you'll get output that can be formatted by a generic TROFF (not even DITROFF is required!). Of course, you won't have the figures, but the layout will be the same, because the space has been reserved.

The default level that *psfig* runs at is 100. In-line figures get a cost of 5 by default, and broken out figures get a cost of 10. To set the cost of a figure, simply tack on a `level` clause. The box above was made by:

```
rosette.ps height 1i level 9999
```

8. Using Psfig

Since *psfig* is a TROFF preprocessor in the classic style, it operates as a pure filter. It can be used anywhere in the pipeline of preprocessors, but it is safest if you run it last (just before TROFF). The macro definitions of `.F+` and `.F-` must be included with the `-mpsfig` option on the DITROFF command line. For example, this paper was produced with the equivalent of:

```
pic | tbl | eqn | psfig |
ditroff -mpsfig | psdit
```

There are a few options that can be specified on the command line.

`-d<level>` specifies the draft level to run at. If `<level>` is omitted, then zero is assumed, causing all figures to be omitted.

`-f` specifies that DITROFF codes should be output that work around a bug in DITROFF that was discovered during the development of *psfig*. Broken out figures won't center properly with unfixed DITROFF's without this flag. Also, special characters in *eqn* won't work on these unfixed DITROFF's, even with `-f`.

`-D<dir>` specifies a directory in which to search for files. Any number of these can be specified, and they will be searched in turn. The current directory is always searched first.

9. Psfig/T_EX

We have a package of similar functionality available for the T_EX document preparation system. *Psfig/T_EX* uses no preprocessor, and is implemented entirely in T_EX macros, using the *dvips* postprocessor from ArborText. Files are scanned for the `%BoundingBox` comment, but they are not checked to see that they conform to the structuring convention (for example, the bounding box could be in the middle of the file, rather than in the header or trailer as required by the convention). The *psfig/T_EX* command

```
\psfig{file=name, clause, clause,... }
```

is the equivalent of the *psfig* command

```
figure name clause clause ...
```

The POSTSCRIPT implementation of *psfig* and *psfig/T_EX* are very similar, differing only in the scaling factors used to convert DITROFF or T_EX units into points, and in the code to implement `global` (each post-processor has different variables that must be restored when we return to the

current save context.)

10. Getting Psfig

Inquiries about *psfig* may be directed to trevor@linc.cis.upenn.edu, or the U.S. mail address listed above. *Psfig* will be available as part of future releases of the TRANSCRIPT package from Adobe, as well as through uucp/ftp distributions.

11. Acknowledgments

We would like to thank: the University of Pennsylvania and in particular Ira Winston for supporting and encouraging this work; Brian Kernighan for helping gracefully with the internals of DITROFF and for having written *eqn* to guide us through the darkness; and Adobe Systems for having designed and implemented POSTSCRIPT, which made it all possible.

A. Language Syntax

Psfig recognizes these commands found between *.F+* and *.F-* or in-line delimiters:

```
[figure] path [clauses] [modifiers]
file path [modifiers]
literal text [modifiers]
space dimen
define word text
delim char[char]
```

Modifier is one or more of:

```
level num
global
```

Clauses is one or more of:

```
height dimen
width dimen
bounds int int int int
reserve dimen dimen
clip
{ enviornment }
```

Environment is a series of file and/or literal commands, and the keyword *figure*.

Path is a valid Unix file path.

Dimen is a TROFF expression that will evaluate to a length.

Text is any string of characters which is delimited by a single character, and does not contain

that character.

The path, clauses, and modifiers of a figure command may be present in any order. Whitespace is ignored (except that in-line commands may not cross lines), and semicolons optionally separate commands.

AN ENVIRONMENT FOR SGML DOCUMENT PREPARATION

*Le van Huu
Department of Computer Science
University of Milan
Via Moretto da Brescia, 9 Milan - Italy*

Abstract

SGML (Standard Generalized Markup Language) is an International Standard defined by ISO for documents description based on generalized markup technique. This paper refers to some proposals for the application of SGML concepts in the formatting environment. It describes an environment for SGML documents preparation, where the user, even inexperienced, is able to define the logical structure and the text of documents interactively and graphically, and where the document so defined can be processed by all formatting systems.

1. Introduction.

Recently, the continual increase of personal computers with their own user-friendly word processors have been leading authors to prefer generating their manuscripts electronically. Generally, in order that a formatter is able to produce the final pages of a document it is necessary that the author describes it by means of additional information inserted into the text. This additional information, called "markup elements", establishes, directly or indirectly, actions to apply upon the text, and forms a real text description language with its own semantic and syntactic rules. A markup language, in line with information that its elements are able to represent, can be inserted into one of the following classes: [FUR82]

- a) Procedural markup, which consists of a set of low-level commands which allow the author to drive and control the formatting process (underline a phrase, change the font...) exactly at the point of the text where the markup is inserted. There are many formatters from in the 1960s in this category, such as FORMAT from IBM [BER69], RUNOFF from M.I.T [SAL65], etc.
- b) Declarative markup, which concentrates on the logical structure of the document objects bringing out their attribute values (figure with the figure caption, first paragraph...) as well as the relationship between one element and the others (a figure inside a paragraph, footnote of the preface...) [LIG79]. In this case, the author has only to specify that a phrase is the title and the other is the publisher's name, rather than it must be underlined or centered. In fact, the choice of processing instructions that affect the various part of the text is left to the formatting system, which must give them a proper layout, according to their sense. Early systems, such as GML [IBM76], SCRIBE [REI80], belong to this category. Some integrated editor/formatters also offer what is essentially a declarative language, such as ETUDE [HAM81], Bravo [FUR82], Star [SMI82].

On the other hand, along with electronic document preparation and publishing, many powerful and sophisticated formatting and typesetting systems have been realized. Thanks to this proliferation, publishers have the opportunity to select from among many possibilities the formatter which suits most their needs. This could cause a certain difficulty for authors who have to prepare the manuscript using the specific markup language established by their publishers.

Instead, if some publishers are not so demanding and accept manuscripts in any format then the problem is reversed: they have to adapt the manuscripts to their own organization, inserting

formatting codes and typesetting information in the manuscripts in accordance with the formatter system available to them. In this process it is not easy to recognize from the manuscript file the text portion belonging to the specific object class (e.g. title of document, items of list, title of appendix) to which to associate the formatting functions.

This difficulty could be diminished if the manuscript has been described using the SGML markup language. SGML (Standard Generalized Markup Language) is an International Standard defined by ISO for documents description. It is based on the generalized markup technique which identifies each element of the document associating with it a logical class [GOL81]. SGML considers a document as an element composed of other elements [SMI85]. The relationships among these elements constitute the document structure. SGML provides a coherent and unambiguous syntax for describing that structure; in other words, it is a formal expression of document markup. Using SGML, processing and formatting instructions are external to the text, which therefore is not dependent on any particular application [ADL85].

This paper presents a system for SGML documents production to operate in the electronic publishing area. The main objectives of the system are: to provide an easy-to-use tool for SGML documents editing, based on their logical structure and capable of relieving authors from typesetting problems; and to allow these documents to be processed by all formatting systems (i.e., it is not necessary for authors to know which formatter will process their documents).

2. SGML.

In this section we will illustrate some of SGML's main features. Although what is reported is not a full description of the language, it is enough to understand concepts which will be laid out in this paper. Other information about SGML is found in the ISO document [ISO86].

SGML considers every document made up of elements which are organized in a hierarchical structure. A book, for example, could contain a "chapter" element, that in turn contains "paragraph" and "pictures" elements; then the "paragraph" element could contain in turn "text", "example", "note" elements. Each of these elements could finally contain character sequences that represent its "content". Every element of the structure is identified by a symbolic name called "generic identifier" (we will refer to it as GI).

SGML supplies the user with markup elements that are divided into two main classes: **declaration markup** and **descriptive markup**.

The former is a set of statements, used principally to establish the characteristics and roles of every element in the document, i.e., its relationship with the other elements. By means of declaration markup, therefore, it is possible to construct the logical structure of the document. Precisely, the declaration for an element consists of specifying its "content model", i.e., relationships among subelements contained in it, and its "attribute model".

The content model represents the order and the number of occurrences of document elements. It is specified using "delimiters", represented by Meta-characters as "|", "&", and put among the subelements. There are two types of delimiters. The first is "connector" and comprehends the following characters:

- " , " : represents the sequential order of the elements;
- " | " : represents the alternative presence of elements: i.e., the presence of one element excludes the presence of all others (OR connector);
- " & " : represents the simultaneity of the elements: all of the connected elements must occur in the document, but in any order (AND connector).

Occurrence indicators constitute the second type of delimiters. They consist in:

- the optional element (represented by "?"): the element can occur zero or one time;
- the required and repeatable element (represented by "+"): the element can occur one or more times;

- the optional and repeatable element (represented by “*”): the element can occur zero or more times.

An example of element declaration could be the following

```
1 <!ELEMENT article (title, author+, (keyword | abstract), paragraph*) >
2 <!ELEMENT title CDATA >
3 <!ELEMENT abstract CDATA >
4 <!ELEMENT paragraph CDATA >
5 <!ELEMENT keyword CDATA >
6 <!ELEMENT author CDATA >
```

They establish that the element “article” contains the element “title”, followed by one or more “author”, which in turn is followed by both the element “keyword” or the element “abstract”. Finally, these elements are followed by 0 or more “paragraph” elements. Moreover, the keyword CDATA indicates that the data type of elements containing it can be constituted by any character.

Associated with the structure declaration of an element there is the attribute declaration, specified by the “attribute model”. Two text portions can be considered different, even if identified with the same GI, when they have two different attribute values. The attribute model of an element, inserted into its declaration, consists of the list of all possible attribute names, their corresponding possible values, and their default values.

The above example can be completed by the following

```
1 <!ELEMENT article (title, author +, (keyword | abstract), paragraph*) >
2 <!ELEMENT (title | abstract | keyword | paragraph) CDATA >
3 <!ELEMENT author CDATA
```

```
4      --ATTRIBUTE      VALUE      DEFAULT--
      type      (principal | coauthor)      principal>
```

where the last line represents an attribute declaration. It indicates that the attribute “type” of the element “author” can assume the value “principal” or “coauthor” and that the default value of “type” is “principal”. Moreover strings delimited by “--” represent comments.

Several attribute names could have the same value model. For example, the following declaration

```
1 <!ELEMENT author CDATA
2   (type | order) (first | second) first>
```

indicates that the attributes “type” and “order” can assume either the value “first” or the value “second”.

The logical structure constructed in this way determines a “document type”, to which several documents can refer. In fact, describing a document means marking text portions by their correspondent names (GIs), i.e., collocating them in the structure established by the document type, according to their logical meaning. This operation is carried out using descriptive markup tags.

A descriptive markup tag comprehends, besides the element name, the attribute specification, which in turn contains the attribute names and their relative values. As mentioned, attribute specifications are useful for distinguishing those text portions which are identified by the same GI.

A very simple example of a complete SGML document could be the following:

```
1 <!DOCTYPE article
1 [<!ELEMENT article (title, author +, ( keyword | abstract ), paragraph *) >
3 <!ELEMENT (title | keyword | abstract | paragraph) (#CDATA) >
4 <!ELEMENT author (#CDATA)
5   type (principal | coauthor) principal> | >
6 <article>
```

```

7 <title>
8   The SGML standard proposed by ISO.
9 <author type = "principal">
10  Le van Huu
11 <author type = "coauthor">
12  E. Terreni
13 <keyword>
14   languages, text structure, standard.
15 <paragraph>
16   This is the first paragraph ...
17 </paragraph>
18 </article>

```

Apart from the first lines representing the markup declaration previously described, the remaining lines identify text elements. They are delimited by descriptive markup tags containing their relative element names (e.g. lines 7, 9, 17 etc.). In particular, lines 9 and 11 report the same GI ("author"), but with two different attribute values, respectively, "principal" and "coauthor". They distinguish in this way the two authors.

3. SGML implementation.

From the examples presented above we can notice that one of the problems using SGML could regard its syntax. SGML has a user-friendly syntax, and, as such, its statements are often too long. Users might easily commit syntax errors when they describe their documents.

Another problem regards the formatting process of the SGML document. As mentioned, using SGML low level processing instructions are external to the text. But if, on one hand, this characteristic enhances the portability of SGML documents, on the other, it poses a potential problem: we can not immediately obtain the final pages of the document after the parsing task; we also need tools with low-level functions capable of resolving typesetting problems [LEV85a].

In this section we will illustrate a SGML implementation that intends to create an environment where solutions the problems considered above are the main objectives to achieve [LEV85b]. The implementation is realized under the UNIX^o O.S. System III, using the C language.

Precisely, the environment includes:

- an interactive document input system based on manipulation of windows, where document elements are represented by boxes designed on screen;
- a translator which generates a sequential file containing document text and structure definitions conforming to SGML notation, using information coming from the previous system. This document constitutes the source file for the SGML parser;
- a SGML parser which is capable of processing documents coming from either the input system mentioned or a normal text editor and of producing an intermediate and system-independent file containing different information relating to the physical structure of the document;
- a map table to associate formatting action with every document element. The idea consists in transforming, with the support of this map table, the SGML document into a source file for every formatter desired, then to submit it to the formatter itself. In other words, the map table contains the association between every document element and the formatting commands of every formatter. The mapping is described by means of a flexible language, constructed specifically for this purpose. It is denominated METAFORM [LET86].

These components will be examined in detail in the next sections.

^o Unix is a trade mark of AT& t Bell Laboratories

3.1 Interactive graphical input.

This component provides the user with the possibility of building document structures, and of inputting document contents interactively. The idea is based on the SGML document's characteristic of having a hierarchical structure. We can imagine representing the structure by designing boxes on the visual display unit. Every box represents a logical element of the document. Positions of boxes determine relationships between the elements that they represent. In this context, the user is able to make:

- Document element declaration: to build logical structures. They are equivalent to the use of SGML declaration markup.
- Document descriptive: to associate text portions with boxes to describe a particular document. This operation is equivalent to SGML descriptive markup specification.

The first functions allow the user to build every document structure easily by moving the cursor and designing boxes on video.

There are some standards relating to box positions:

- two boxes designed one over the other represent document elements that are in sequence order, i.e., separated by a "sequence delimiter" (comma);
- boxes aligned horizontally represent either elements that are mutually exclusive ("|" delimiter) or elements that have to be present simultaneously ("&" delimiter);
- boxes contained inside another represent elements at a lower level of the document structure; i.e., they are the content model of an element represented by the outer box;

Every box represents a document element, therefore it is necessary to associate with it information about the name and the occurrence of the element itself. In fact, every time the user terminates the creation of a box, the system asks him/her questions about the name of the GI to associate with the box, as well as its occurrence indicator type, which can be, as mentioned, the following characters: "?", "*", "+". The occurrence indicator and the GI name specified by the user are displayed on the box concerned.

Moreover, if the user designs two boxes on the same horizontal line, he/she intends to represent two elements at same level of the document structure. Therefore, these elements have to be connected together by either the connector "&" (AND) or the connector "|" (OR). For this purpose the system will pose to the user the following question:

Connector ? (| , &)

and the character chosen by the user will represent the connector of these elements.

Furthermore, the user is provided with functions to move, copy, wider, contract and cancel boxes interactively in order to build any document logical structure he/she desires.

The Figure 1 refers to an example representing boxes designed on video which constitute the document structure of "article" and the corresponding SGML element declaration.

Besides the construction of elements content model, the user can specify the attribute model of an element at any time simply by selecting this option from the menu. Then he/she can move the cursor through the screen and point to the box that represents the element with which he/she will associate attribute information. Under normal conditions, there are no signs of the presence of attributes for a box; but with a simple command the user can obtain a screen representation in which boxes with attribute values are marked by special graphics.

Selecting a certain box, the user is able to read and update all information on the attributes of the element concerned. This information will be displayed little by little by type, in the following order. First, a window for the attribute name group will be opened. The user can insert and modify every name. For example, the window "a" of Figure 2, which refers to the attribute model

of "aut" element reports two attribute names: "type" and "order". Once the attribute names have been specified, another window representing attribute values will be opened next to the previous one, reporting a list of possible standard attribute values (e.g. ID, IDREF ...). The user can both select one of them to associate with the attribute concerned or insert other values, e.g., "first" and "second" (window "b" of Figure 2). Similar operations are repeated for the specification of attribute default value (window "c" of Figure 2).

STRUCTURE CREATE

title

aut*

keys & abstract

article

```
<! ELEMENT article (title, aut* , (keys & abstract) ) >
```

Fig. 1

ATTRIBUTE CREATE

title

aut*

NAME	VALUE	DEFAULT
type	first	first
order	second	first

abstract

```
<! ELEMENT article (title, aut* , (keys & abstract) )
aut CDATA
(type | order) (first | second) first >
```

Fig. 2

Once a document logical structure (i.e. document type) is established, it can be stored in a proper library, to which several documents can refer. The user is able to select a particular document type (e.g. article, manual ...) from the library and to associate text contents with its elements. This procedure is equivalent to the document description using SGML descriptive markup. When boxes representing the selected document structure are displayed on screen, the user can perform this task in the following way:

- 1) pointing the cursor on the box into which he/she wants to input element content
- 2) the system opens three windows relating to attributes information of the element selected. Moving the cursor in these windows, the user can select the attribute names and the corresponding values to associate with the text portion that he/she will input.
- 3) once terminated the attribute values specification, the user is able to input text portion by means of a common text editor just activated by the system. The text portion created will be the content value of the element selected.

The graphical input system described uses principally a Window Manager package realized at the University of Milan [FRO84]. It is similar to the Curses package, but it comprehends new data structures and functions for the manipulation of overlapped windows. Precisely, the package considers windows on the screen as transparencies laid one upon the other, rather than as opaque sheets placed on a desk. This possibility is useful for our input system because we need a box not to hide windows contained inside it, even when the box itself is the current one. New data structures are created in order to allow refresh operations to realize this functionality in an optimal way.

3.2 The translator.

Structures and texts produced during an interactive graphical input session are not controlled directly by the SGML parser. In fact, an intermediate module (called "translator") will translate the effect of graphical input operations to a file with SGML syntax, as if it had been created with an ordinary text editor. This file will be the source file for the SGML parser.

This schema has been chosen because we prefer to make the SGML parser capable of processing even documents generated by environments lacking in the interactive graphical input system described above.

3.3. The SGML parser.

SGML documents, either generated by an ordinary text editor or coming from the interactive document input system, are processed by the SGML parser. Its main task is interpreting tags and element relationships, as well as attribute specifications, according to document type definition. Other SGML elements which are not presented in this paper, such as entities reference, marked sections ..., are also recognized and controlled by the SGML parser, which in turn produces an intermediate file containing several pieces of information relating to the physical structure of the document [LEV86]. The main pieces of information contained in the intermediate file for every GI of the document are:

- a) name of GI
- b) name and value of attributes associated
- c) pointers to the relating text portion in the SGML source file.

This file will be useful during the formatting process, as we will see shortly.

The SGML parser is realized with the support of Lex [LES75] and Yacc [JOH75] tools. Data structures of the parser are different from those of a common compiler because we need complex and particular checks on document structures.

3.4. The formatting process.

Since the style of presentation is not present in a SGML document, to provide its final form it is necessary to associate logical elements of the document with formatting actions. This could be achieved either by creating for every document type a particular "profile" which specifies the association between document elements and formatting procedures deliberately constructed (this approach is used for example by the SCRIBE system, and GML of IBM), or by transforming the SGML document into the source file for every formatter desired, then to submit it to the formatter itself [LEV85b].

The second solution is probably more interesting in the publishing environment because it takes into account a particular need of publishers: to be able to use their own formatters to produce the final document. With this approach the transformation from the SGML document into the source file for a formatter could be performed by means of a particular map table that associates every logical element with commands (control sequences) of the formatter selected.

But not always a static association between a document element (e.g. title) and a fixed set of formatting commands (e.g. .TL, .sp for NROFF system [OSS76]) is possible. It may happen, for example, that the same element "title" could require two different treatments, according to whether it belongs to the appendix or to the chapter of the document. In other words, the processing actions applied to the same element could change according to its position in the document. Furthermore, since SGML is a descriptive language with its variables and parameters (e.g. attribute values), the mapping must depend on the current global state of these SGML objects.

Therefore, the map table should be described using some flexible language which, at run time, is able to select the proper set of formatting commands to be inserted into the source file on the basis of the current status of every element in the document. We have defined to this end a language, called METAFORM, which is capable of expressing, by means of control structures and predefined procedures, the processing to associate with various SGML markup elements present in a manuscript.

Every METAFORM program refers to a specific formatter and to a specific document type. Logically, a METAFORM program is divided into blocks containing instructions for GIs. Every GI block, in turn, is divided into blocks representing instructions for its attributes. As we will see shortly, every time that a GI or an attribute is matched in the document, the related METAFORM instructions block is executed. A METAFORM program must be compiled by the METAFORM compiler which then produces a set of pseudo-code (p-codes) of a hypothetical stack machine [WIR81]. By means of METAFORM compiled programs, the formatting process of a SGML document is the following.

Once decided which formatter will process the document and, consequently, which METAFORM program is to be executed, the intermediate file of the document under examination is scanned by a proper p-codes interpreter. When a GI occurs, the interpreter searches forward for the related attributes. For every attribute matched, the set of the corresponding METAFORM instructions, (or, better yet, the p-codes generated), is executed, just as in the case of a procedure call. METAFORM I/O instructions usually access to the SGML source document, by means of pointers present in the intermediate file, to retrieve texts which are to be worked on. Execution of METAFORM instructions for an attribute will produce as output a part of the source file for the formatter selected. At the end of the intermediate file scanning, the entire source file is generated. Then it is submitted to the formatter, which will compose it generating the final document.

A very simple METAFORM program, which refers to the TeX formatter [KNU79], is reported here as an example.

```
01 BEGIN_GI author
02   BEGIN_ATT type
03     WRITE ("\\centerline {");
04     READTRANSFER (); -- author name
05     WRITE ("}");
06     IF ($ATTVAL == "coauthor") THEN
07       BEGIN
```

```

08             WRITE ("\\vskip 3.50pt");
09         END
10     ELSE
11         BEGIN
12             WRITE ("\\vskip 2.50pt");
13         END
14     END.ATT
15 END.GI

```

The example shows some elements of the language. For instance, \$ATTVAL is a system variable representing the value of the attribute just matched in the intermediate file; WRITE is a procedure which produces in output the string specified as argument; READTRANSFER gets the whole text portion of the GI from the SGML document and copies it on the output file without making any modifications.

Every time the GI "author" occurs in the document with the attribute "type", the program will produce (by means of the WRITE and READ instructions reported in lines 3, 4 and 5) parts of the output file in the following order: first, the string " \\centerline { ", then the author name (retrieved from the SGML source document) and, finally, the character " } ". Moreover, if the value of the attribute "type" is "coauthor" the program will write the string "\\vskip 3.50pt"; otherwise, "\\vskip 2.50pt" will be produced.

From what we have described about the formatting process, it seems that the most complicated task is the METAFORM program construction. But we must not forget that this work is performed just once for all possible SGML documents that follow the same logical structure. Once the map table is constructed, the METAFORM executable codes are able to interpret whatever document the user desires. Moreover, despite the fact that METAFORM is a programming language, it is not oriented to expert programmers. It is designed for persons who construct the map table defining the output format of documents. These persons, experts in composing and typesetting problems, are not necessarily familiar with all programming techniques. Therefore, the language tries to be as easy and natural to use as possible, even if this makes the language not very powerful. For this reason, METAFORM is provided with a very small set of statements. Instead, we prefer to define several useful standard procedures and functions and system variables so as to make the work of the programmer easier.

As for the SGML parser, we have realized the METAFORM compiler and the p-codes interpreter using Lex and Yacc packages.

4. Conclusions.

The system for SGML document production described in this paper offers some advantages for authors and publishers. In regard to the authors, they are relieved of typesetting problems; therefore, they are able to concentrate their attention on the content of their manuscript. These advantages derive from SGML features. In the environment proposed authors could transmit to publishers the SGML source documents, or, more probably, the intermediate file, if they have on hand a SGML parser.

These files could be transmitted to all publishers, independently of the kind of formatting program they have on hand. Publishers have only to execute the interpreter module to process the intermediate file received and to generate the source file for their formatter. If the document that they receive does not correspond to any structure of GIs present in the map table, the work of the publishers increases somewhat: they have to add processing instructions in the map table using METAFORM language.

References.

- [ADL85] S. Adler, B. Davis: "SGML tutorial", GenCode/SGML Orientation Tutorial (Heidelberg, Germany 3 June 1985).
- [BER69] G.M. Berns: "Description of FORMAT, a text processing program", Comm. ACM Vol. 12, N.3 March 1969.
- [FUR82] R. Furuta, J. Scofield, A. Shaw: "Document Formatting System: Survey, Concepts and Issues", ACM Computing Survey, Vol. 14, N. 3, September 1982.
- [FRO84] M. Frontini: "Interfaccia utente orientata ad una gestione multiwindow", M.S. dissertation, Computer Science Dep., Univ. of Milan, 1984.
- [GOL81] C.F. Goldfarb: "A generalized approach to document markup" in Proc. ACM SIGPLAN/SIGOA Conference Text Manipulation (Portland, Ore., June 8-10 1981), ACM, NY, 1981.
- [HAM81] M. Hammer et. al.: "The implementation of ETUDE, an interpreted and interactive document production system", in Proc. ACM SIGPLAN/SIGOA Conference Text Manipulation (Portland, Ore., June 8-10 1981), ACM, NY, 1981.
- [IBM76] INTERNATIONAL BUSINESS MACHINES: "Document Composition Facility: GML Quit Reference Summary", IBM Data Processing Division, White Plains, NY, 1976. Order N. SX26-3719-0.
- [ISO86] ISO: "Information Processing - Text and Office Systems - Standard Generalized Markup language (SGML)", ISO 8879 - 1986 (E).
- [JOH75] Stephen C. Johnson: "YACC Yet Another Compiler-Compiler", CSTR Report N32, Bell Laboratories 1975.
- [KNU79] D.E. Knuth: "TeX and METAFONT: New Directions in Typesetting", Digital Press and the American Mathematical Society, Bedford, Mass, and Providence, R.I. 1979.
- [LEV85a] Le van Huu: "TeX and ISO/STPL Standard" in Proc. of the First European Conference on TeX for Scientific Documentation (Como, Italy 16-17 May 1985). Ed. D. Lucarella, Addison-Wesley Publishing, August 1985.
- [LEV85b] Le van Huu: "SGML: A standard language for text description" in Proc. of the Second Intern. Conference on Text Processing Systems (Dublin, Ireland 23-25 Oct 1985). Ed. J.H. Miller Boole Press Ltd. Ireland.
- [LES75] M.E. Lest, E. Schmidt: "Lex - A Lexical Analyzer Generator", Bell Laboratories 1975.
- [LET86] Le van Huu, E. Terreni: "A language to describe formatting directives for SGML documents", in Proc. of "TeX for Scientific Documentation Conference, June 86, Strasbourg, France. Ed. by J. D[sarm]nien, Springer-Verlag.
- [LIG79] C. Lightfoot: "Draft GenCode : Generic textual element identification - A primer", Graphic Communications Computer Association Arlington, 1979.
- [OSS76] J.F. Ossanna: "NROFF/TROFF user's manual", Computer Science Tech. Rep. 54, Bell Laboratories, Murray Hill, N.J., Oct. 1976.
- [REI80] B.K. Reid : "SCRIBE: A document specification language and its compiler", Ph. D. dissertation, Computer Science Dep. Carnegie-Mellon Univ., Pittsburgh, Pa., Oct 1980.
- [SAL65] J. Saltzer: "Manuscript typing and editing: TYPSET, RUNOFF", in The Compatible Time-Sharing System: A programmer's guide, M.I.T. Press Cambridge, Mass, 1965.
- [SMI82] D.C. Smith, C. Irby, R. Kimball, B. Verplank: "Designing the Star user Interface", Byte 7, 4 (April 1982).
- [SMI85] Joan M. Smith: "The computer and Publishing: an opportunity for new methodology" in Proc. of the Second Intern. Conference on Text Processing Systems (Dublin, Ireland 23-25 Oct 1985). Ed. J.H. Miller Boole Press Ltd. Ireland.
- [WIR81] N. Wirth: "Pascal-S: A subset and its implementation" in Pascal-The language and its implementation. Ed. D.W. Barron, John Wiley & Sons, Ltd 1981.

A Unix Interface for Shared Memory and Memory Mapped Files Under Mach

Avadis Tevanian, Jr., Richard F. Rashid, Michael W. Young,
David B. Golub, Mary R. Thompson, William Bolosky and Richard Sanzi

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes an approach to Unix shared memory and memory mapped files currently in use at CMU under the Mach Operating System. It describes the rationale for Mach's memory sharing and file mapping primitives as well as their impact on other system components and on overall performance.

1. Introduction

The 4.2 BSD mapped file interface (*mmap*) was designed to address two shortcomings of previous Unix systems: a lack of shared memory between processes and the need to simplify processing of file data. Early Unix systems had provided no shared memory access and a stylized way of accessing sequential file data through *read* and *write* system calls. Applications that desired random access to data would use Unix's *seek* operation or buffer their data themselves, often incurring unwanted system overhead. A mapped file facility could allow a user to treat file data as normal memory without regard to buffering or concerns about sequential versus random access. It would also provide an obvious mechanism for sharing memory by allowing more than one process to map a file read/write simultaneously.

The BSD file mapping facility was proposed as early as 1982. Since then, similar mapped file interfaces have been implemented by several vendors, both as part of 4.2 BSD Unix (e.g., Sequent Dynix [2]) and as part of a System V modified to contain 4.2 BSD enhancements (e.g., IBM's AIX). A shared memory facility not based on mapped files is available in AT&T's System V and has also been adapted to a variety of 4.2 BSD based systems such as DEC's Ultrix. There is currently a lively debate going on within the Unix community about the appropriate Unix interface to virtual memory and the relationship between mapped files, memory sharing and other virtual memory concerns such as copy-on-write memory mapping.

This paper describes the somewhat atypical approach to shared memory and file mapping currently in use at CMU under the Mach Operating System. It describes the rationale for Mach's memory sharing and file mapping primitives, their impact on other system components and on overall performance and the experiences of the Mach group in implementing and using them.

2. The Problems of a Mapped File Interface

Despite its obvious convenience, the notion that all memory sharing should be channeled through a mapped file interface presents a number of problems:

1. A Unix file is (in principle) a permanent on-disk data structure which must be maintained consistent against crashes. The use of disk files to exchange temporary data can put an unnecessary I/O load on the system and impact performance.
2. A mapped file facility must take into account the sharing of remote (network) files. In order to handle remote file systems (e.g. SUN NFS), the operating system must be intimately involved in maintaining network data consistency. This can increase its complexity considerably by introducing within the OS kernel many of the same concerns that complicate transaction processing systems.
3. Sharing semantics are limited to those supplied by the kernel. In particular, an application program cannot use domain specific knowledge to allow less than full consistency in sharing access to file data. This can result in inefficiency in the handling of data sharing across node boundaries.

These problems typically have led to compromises in the actual mapped file semantics provided. Most have either assumed that modifications to read/write mapped files are not guaranteed to be consistent in the face of multiple writers, or they guarantee consistency only for those processes which share files on a single network node.

3. The Uses of Shared Memory

Many of the potential uses of shared memory do not require a file mapping interface. In fact, such an interface may present problems. Memory sharing is often suggested as a way of overcoming traditional Unix deficiencies by providing for:

- fine granularity multiprocessing,
- ultra-fast IPC,
- database management support and/or
- reduced overhead file management.

But of these potential uses of shared memory, only two require some kind of mapped file facility and of these only one fits the traditional *mmap* model of file access and shared data consistency.

3.1. Fine grain multiprocessing

The need to support fine grain multiprocessing has forced several multiprocessor manufacturers to adopt some form of memory sharing in their multiprocessor versions of Unix, e.g. in Sequent's Dynix [2] and Encore's UMax [3]. This kind of shared memory often takes the form of an *mmap*-like primitive which acts on a special device or file. Fine grain multiprocessing is accomplished by creating as many processes as there are available processors which then *mmap* the shared memory object and synchronize through it. The only reason such a facility would want a mapped file interface is the benefit provided by using Unix's filesystem name space for referring to shared data. There is no need for the shared data to be disk resident or permanent.

3.2. Fast IPC

Shared memory can be used as a kind of ultra-fast IPC facility, especially where large data structures are built in shared memory by one process and then managed or manipulated by another. An example of a potential use of this kind can be found in the relationship between multiphase program components such as the typical C language preprocessor and compiler. Already such programs use files or pipes to accomplish their goals. The advantages of such a fast IPC facility are actually diminished by tying it to a similar shared file construct which would require some form of file system creation/destruction cost as well as disk I/O.

3.3. Database management

Designers of database management systems have argued against Unix at least partly because of its inability to share data between potential database client programs and transaction managers, data managers and recovery logs. Systems of this sort need both sharing between processes and sharing of data pages in files to accomplish their ends. Unfortunately, an *mmap*-like construct does not, by itself, resolve the problems posed by database systems. For example, it may be important for a database system to know when data is going to be moved from volatile storage to disk so that a database recovery manager can update crucial portions of the recovery log in advance [7] (i.e., write-ahead logging). In addition, the consistency of shared memory must either be absolute, or the consistency model must be well understood and manageable by the database transaction manager -- a fact often remarked by database builders on other systems with shared file constructs such as Apollo's Aegis [4].

3.4. Efficient file access

By far the most compelling general argument for linking shared memory with memory mapped files is the need in Unix for reducing the overhead of file management. Partly because Unix was originally designed at a time when primary memory was a scarce commodity, traditional Unix programs are I/O intensive. Even the Unix pipe facility was once implemented as file I/O to conserve memory. As the relationship between the costs of memory and secondary storage have changed, large memory Unix systems are limited more by their I/O capacity than by memory. A mapped file facility could reduce the cost of I/O operations by eliminating a copy operation from the Unix buffer cache to process memory and also provide for better memory utilization by allowing more than one process to share the same physical memory when accessing the same file.

4. Mach Memory Primitives

Rather than support sharing only through an *mmap* model of shared memory through shared files, Mach provides a number of non-file based mechanisms for sharing data among computational entities:

- Unrestricted, fine grain sharing between processors in a tightly coupled multiprocessor can be achieved by using the Mach notion of *thread*. A thread can be

thought of as a lightweight process which shares an address space with other lightweight processes. The Unix notion of *process* has been split into *task* and *thread*. A task defines an address space and resource domain in which a number of program control flows (threads) may coexist. Using this multiple thread per task mechanism, an application may easily share a single address space among separate executing entities.

- In addition to unrestricted sharing using threads, Mach allows tasks to read/write share protected ranges of virtual addresses through inheritance. A Mach task can specify any portion of its address space to be shared read/write with its children as the result of a *task_create* operation. The fact that memory is shared only through inheritance guarantees that the shared memory is always located within a single host (or cluster within a host). This allows the kernel to guarantee cache consistency for such memory. Another advantage of this method of data sharing is that it ensures that shared memory is always located at the same virtual address in each inheriting task. This avoids the often difficult programming problems caused by pointer address aliasing in shared data structures.
- Physical memory can be shared copy-on-write by taking advantage of Mach's integration of IPC and virtual memory management. Applications not requiring read/write memory sharing can use this feature to transfer large amounts of data between tasks without actually copying data. In effect, a multiphase application can effectively forward between components the actual physical memory containing important data. The sender in such an exchange is always protected because data is logically sent by value. The kernel uses memory management tricks to make sure that the same physical page is available to both sender and receiver unless or until a write operation occurs.
- Finally, applications may define their own sharing semantics within a distributed system of Mach hosts using the Mach *external pager* facility. This external pager mechanism allows an application to control many aspects of virtual memory management for regions of virtual memory. An *external pager* may implement fully coherent network shared memory, or a shared memory paradigm that requires clients to maintain their own cache consistency (if consistency is even desired). It allows a database recovery manager to be advised of the kernel's need to flush data to disk in advance and thus permit efficient write-ahead logging.

4.1. Mach virtual memory operations

Table 4-1 lists the set of operations that can be performed on the virtual address space of a task. Mach calls are specified to act on object handles called *ports* which are simplex communication channels on which messages are sent. A more complete description of Mach ports and calling conventions can be found in [1].

A task address space consists of an ordered collection of mappings to memory objects; all threads within a task share access to that address space. A Mach *memory object* (also called a *paging object*) is a data repository, provided and managed by a server. The size of an address space is limited only by the addressing restrictions of the underlying hardware. For example, an IBM RT PC task can address a full 4 gigabytes of memory under Mach, while the VAX architecture allows at most 2 gigabytes of user address space.

The basic memory operations permit both copy-on-write and read/write sharing of memory

Virtual Memory Operations

<code>vm_allocate (task, address, size, anywhere)</code>	<i>Allocate and fill with zeros new virtual memory either anywhere or at a specified address on demand.</i>
<code>vm_copy (task, src_addr, count, dst_addr)</code>	<i>Virtually copy a range of memory from one address to another.</i>
<code>vm_deallocate (task, address, size)</code>	<i>Deallocate a range of addresses, i.e. make them no longer valid.</i>
<code>vm_inherit (task, address, size, inheritance)</code>	<i>Set the inheritance attribute of an address range.</i>
<code>vm_protect (task, address, size, set_max, protection)</code>	<i>Set the protection attribute of an address range.</i>
<code>vm_read (task, address, size, data, data_count)</code>	<i>Read the contents of a region of a task's address space.</i>
<code>vm_regions (task, address, size, elements, elements_count)</code>	<i>Return description of specified region of task's address space.</i>
<code>vm_statistics (task, vm_stats)</code>	<i>Return statistics about the use of memory by task.</i>
<code>vm_write (task, address, count, data, data_count)</code>	<i>Write the contents of a region of a task's address space.</i>

Table 4-1:

All VM operations apply to a *task* (represented by a port) and all but `vm_statistics` specify an *address* and *size* in bytes. *anywhere* is a boolean which indicates whether or not a `vm_allocate` allocates memory anywhere or at a location specified by address.

regions between tasks. Copy-on-write sharing between unrelated tasks is usually the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed. Read/write shared memory within a task creation tree can be created by allocating a memory region and setting its inheritance attribute. Subsequently created child tasks share the memory of their parent according to its inheritance value. The only restriction imposed by Mach on the nature of the regions that may be specified for virtual memory operations is that they must be aligned on system page boundaries. The system page size is a boot time parameter and can be any power of two that is a multiple of the hardware page size.

4.2. Managing external pagers

The basic task virtual memory operations allow memory sharing through inheritance between tasks in the same task creation subtree. Read/write shared memory between unrelated tasks can be implemented through the use of external pagers -- tasks which allocate and manage secondary storage objects.

The Mach interface for external pagers can best be thought of as a message protocol used by a pager and the kernel to communicate with each other about the contents of a memory object. The

external pager interface to the kernel can be described in terms of operations requested by the kernel (messages sent to a *paging_object* port) and calls made by the external pager on the kernel (messages sent to the kernel's *pager_request_port* associated with a memory object). Tables 4-2 and 4-3 describe these two interfaces.

Kernel to External Pager Interface

pager_init (paging_object, pager_request_port, pager_name)	<i>Initialize a memory object.</i>
pager_data_request (paging_object, pager_request_port, offset, length, desired_access)	<i>Requests data from an external pager.</i>
pager_data_write (paging_object, offset data, data_count)	<i>Writes data back to a memory object.</i>
pager_data_unlock (paging_object, pager_request_port, offset, length, desired_access)	<i>Requests that data be unlocked.</i>
pager_create (old_paging_object, new_paging_object, new_request_port, new_name)	<i>Accept ownership of a memory object.</i>
<p>Table 4-2:</p> <p>Calls made by Mach kernel to a task providing external paging service for a memory object.</p>	

A memory object may be mapped into the address space of a task by exercising the *vm_allocate_with_pager* primitive, specifying a paging object port. This port will then be used by the kernel to refer to that object. A single memory object may be mapped more than once (possibly in different tasks). The Mach kernel provides consistent shared memory access to all mappings of the same memory object on the same uniprocessor or multiprocessor. The role of the kernel in paging is primarily that of a physical page cache manager for objects.

When asked to map a memory object for the first time, the kernel responds by making a *pager_init* call on the paging object port. Included in this message are:

- a *pager_request* port, which the pager may use to make cache management requests of the Mach kernel,
- a *pager_name* port, which the kernel will use to identify this memory object to other tasks in *vm_regions* calls.¹

The Mach kernel holds send rights to the paging object port, and send, receive, and ownership rights on the paging request and paging name ports.

In order to fulfill a cache miss (i.e. page fault), the kernel issues a *pager_data_request* call specifying the range (usually a single page) desired. The pager is expected to supply the

¹The paging object and request ports cannot be used for this purpose, as access to those ports allows complete access to the data and management functions.

requested data using the *pager_data_provided* call on the specified paging request port. To flush modified cached data, the kernel performs a *pager_data_write* call, including the data to be written and its location in the memory object. When the pager no longer needs the data (e.g. it has been successfully written to secondary storage), it is expected to use the *vm_deallocate* call to release the cache resources.

Since the pager may have external constraints on the consistency of its memory object, the Mach interface provides some functions to control caching; these calls are made using the pager request port provided at initialization time.

External Pager to Kernel Interface

vm_allocate_with_pager (task, address, size, anywhere, paging_object, offset)	<i>Allocate a region of memory at specified address backed by a memory object.</i>
pager_data_provided (paging_object_request, offset, data, data_count, lock_value)	<i>Supplies the kernel with the data contents of a region of a memory object.</i>
pager_data_lock (paging_object_request, offset, length, lock_value)	<i>Prevents further access to the specified data until an unlock.</i>
pager_flush_request (paging_object_request, offset, length)	<i>Forces physically cached data to be destroyed.</i>
pager_clean_request (paging_object_request, offset, length)	<i>Forces modified physically cached data to be written back to a memory object.</i>
pager_cache (paging_object_request, should_cache_object)	<i>Notifies the kernel that it should retain knowledge about the memory object even after all references to it have been removed.</i>
pager_data_unavailable (paging_object_request, offset, size)	<i>Notifies kernel that no data is available for that region of a memory object.</i>

Table 4-3:
Calls made by a task on the kernel to allocate and and manage a memory object.

A *pager_flush_request* call causes the kernel to invalidate its cached copy of the data in question, writing back modifications if necessary. A *pager_clean_request* call asks the kernel to write back modifications, but allows the kernel to continue to use the cached data. A pager may restrict the use of cached data by issuing a *pager_data_lock* request, specifying the types of access (of read, write, execute) which may be permitted. For example, a pager may wish to temporarily allow read-only access to cached data. The locking on a page may later be changed as deemed necessary by the pager.

When a user task requires greater access to cached data (e.g. a write fault on a read-only page) than the pager has permitted, the kernel issues a *pager_data_unlock* call. The pager is expected to respond by changing the locking on that data when it is able to do so.

When no references to a memory object remain, and all modifications have been written back to the paging object port, the kernel deallocates its rights to the three ports associated with that

memory object. The pager receives notification of the death of the request and name ports, at which time it can perform appropriate shutdown.

In order to attain better cache performance, a pager may permit the data for a memory object to be cached even after all address map references are gone by calling *pager_cache*. Permitting such caching is in no way binding; the kernel may choose to relinquish its access to the memory object ports as it deems necessary for its cache management.

The Mach kernel may itself need to create memory objects, either to provide backing storage for zero-filled memory (*vm_allocate*), or to implement virtual copy operations. These memory objects are managed by a *default pager* task, which is known to the kernel at system initialization time. When the kernel creates such a memory object, it performs a *pager_create* call (on the default pager port); this call is similar in form to *pager_init*. Since these kernel-created objects have no initial memory, the default pager may not have data to provide in response to a request. In this case, it should perform a *pager_data_unavailable* call.

Since interaction with pagers is conducted only through ports, it is possible to map the same memory object into tasks on different hosts in a distributed system. While each kernel keeps its own uses of the cached data consistent, the pager is responsible for any further coordination. Since each Mach kernel will perform a *pager_init* call upon its first use of a memory object, including its own request and name ports, a pager can easily distinguish the various uses of its data.

5. A Unix Interface for File Mapping

Shared memory can be obtained in Mach either through the use of memory inheritance or external pagers. Given these mechanisms for sharing data, there is no need to overload the Unix filesystem in order to provide shared memory. Nevertheless, the potential performance advantages of mapped files make them desirable for Unix emulation under Mach. In addition, the ease of programming associated with mapped files is attractive in both the Unix and Mach environments.

At present, Mach provides a single new Unix domain system call for file mapping:

```
map_fd(fd, offset, addr, find_space, numbytes)
      int          fd;
      vm_offset_t  offset;
      vm_offset_t  *addr;
      boolean_t    find_space;
      vm_size_t    numbytes;
```

Map_fd is called with an open Unix file descriptor (*fd*) and if successful results in a virtual copy of the file mapped into the address space of the calling Unix process. *Offset* is the byte offset within the file at which mapping is to begin. The offset may be any byte offset in the file, page alignment is not required. *Addr* is a pointer to the address in the address space of the calling

process at which the mapped file should start. This address, unlike the offset, must be paged aligned. If *find_space* is TRUE, the kernel will select an unused address range and return it in **addr*. The number of bytes to be mapped is specified by *numbytes*.

The implementation of *map_fd* was a straightforward application of internal Mach primitives for virtual copying regions of memory and external pagers [5, 8]. When a request is made for a file to be mapped into a user address space, the kernel creates a temporary internal address space into which the file is mapped. This mapping is accomplished with the *vm_allocate_with_pager* primitive. The kernel specifies that new memory is to be allocated and that the new memory will be backed by the internal kernel *inode pager*. Then the file data is moved to the process address space by a call to *vm_copy*. Once this is done, the kernel can deallocate the temporary map.

6. Uses of Mapped Files in Mach

Files mapped using *map_fd* can be used in a variety of ways. Mach itself uses file mapping internally to implement program loading. File mapping can also be used as a replacement for buffer management in the standard I/O library.

6.1. File Mapping and Shared Libraries

Mach uses the mapped file interface to implement both program loading and a general form of shared libraries. In the current Mach system, there are two types of program loaders. The first program loader executes in the kernel and implements the Unix *exec* system call. This loader handles both *a.out* and *COFF* format binary files for binary compatibility with existing systems. The second loader executes in a user task and handles *MACH-O* format binary files. Both loaders use mapped files.

The MACH-O format was devised to be flexible enough to be used as a single file format for fully resolved binaries, unresolved object files, shared libraries and "core" files. It provides enough backward compatibility with older formats (e.g., *a.out*) to salvage most existing code for debuggers and related applications.

The MACH-O format can roughly be thought of as a sequence of commands to be executed by a program loader. The layout of a MACH-O file is summarized as:

```
start
    header
    command_id, command_info
    command_id, command_info
    .
    .
    .
    command_id, command_info
ENDMARKER
```

Each command consists of a command identifier followed by a command-dependent number of

arguments. Some of the commands supported are:

READ_ONLY	Map in data read-only (e.g. a text segment).
WRITEABLE	Map in data read/write (e.g. a data segment).
ZEROFILL	Allocate zero-fill memory (e.g. a bss segment).
REGISTER	Create a thread in the task and set its register state.
LOADFILE	Map in data from another file (e.g. a shared library).
RELOCATE	Relocate a specified address.
END_LOAD	Loading complete.

The *header* contains a magic number indicating MACH-O format. It also contains other useful information such as version information, and a machine-type specifier. Finally, the header specifies the type of file represented, e.g. executable, object file or shared library.

The MACH-O program loader operates by scanning a load file and executing commands as necessary. In the typical case, it uses the *map_fd* call to map portions of files into its address space. It then places the data in the image to be executed using the *vm_write* operation. Since copy-on-write is used at the base of the virtual memory primitives it is possible to share both code and writable data. Each task that writes data within a shared library will get a new copy as each page is written for the first time. Pages that are not written will be physically shared by all tasks.

6.2. File Mapping and Standard I/O

The Mach mapped file mechanism has been used to build a new version of the C library buffered I/O package. When a file is *fopened* it is mapped in its entirety into the caller's address space. The semantics of the buffered i/o package are not changed. The existing stdio buffer has, in effect, been enlarged to the size of the file. When a write takes place only the data buffer is changed. The file is not guaranteed to change on disk until a *fflush* or *fclose* takes place. As with normal buffered I/O, if two processes have the same file open for reading and writing, there is no guarantee how the reads and writes will intermix. A read may get new information off the disk copy of the file, or it may use information that was already buffered.

The primary rationale for this change is improved performance. Table 6-1 shows the time for simple buffered I/O operations both with and without the change. In addition to improved performance, the use of file mapping also has the effect of reducing the memory load on the system. In a traditional Unix implementation *fopen* would allocate new memory to the calling process and copy the data from the Unix buffer cache into that new memory at the time of a *read*. Using this new package and Mach file mapping, each new call to *fopen* will reuse any physical memory containing file data pages, reducing the number of I/O operations. (See table 6-2.)

In addition to traditional buffered I/O calls, the mapped file version of buffered I/O has had added to it a new call which allows an application program to directly access the mapped file data and thus further improve performance by eliminating the copying of data by *fread* and *fwrite*.

Unmapped vs. Mapped Buffered I/O Performance				
Test program	First time user system elapsed		Second time user system elapsed	
old_read	6.1u	0.62s 0:08	6.1u 0.62s	0:08
new_read	6.0u	0.71s 0:08	6.0u 0.21s	0:06
map_read	2.8u	0.76s 0:04	2.7u 0.17s	0:03

Table 6-1:
Time to read a 492544 byte file using standard I/O.
(Mach, 4K file system, MicroVAX II)

old_read performs *fopen* followed by a loop of *getc* calls.
new_read is identical to old_read with new mapped file package.
map_read uses *fmap* and reads data by array reference, not *getc*.

Multiple Access File I/O Performance				
Test program	user	system	elapsed	I/O
old_read[1]	25.4u	1.8s	1:23	217io
old_read[2]	25.3u	2.0s	1:26	326io
old_read[3]	25.1u	2.2s	1:26	439io
new_read[1]	24.0u	1.6s	1:17	89io
new_read[2]	24.0u	1.8s	1:18	194io
new_read[3]	24.2u	1.6s	1:18	197io

Table 6-2:
Time to read a 1970176 byte file using standard I/O.
(Mach, 4K file system, MicroVAX II)

Each program is run 3 times in parallel and times are listed.
Instance numbers for each invocation are in brackets.
Each program accesses the same file simultaneously.

old_read performs *fopen* followed by a loop of *getc* calls.
new_read is identical to old_read with new mapped file package.

The new routine is called *fmap* and is a buffered I/O compatible version of *map_fd*.

To read map a file with *fmap* the user calls:

```
stream = fopen("filename", "r"); /* existing call */
data = fmap(stream, size);      /* new call */
```

where *data* is a pointer to a region of virtual memory where the file contents are buffered, and *size* is the suggested size for the data buffer; if that size is zero, then the implementation will

choose a suitable size. As before,

```
bufsize = fbufsize(stream)
```

returns the actual size of the buffer. Once *fmap* is called, the user can reference file data by using the data pointer and any offset less than *bufsize*. The user may also mix *fseek*, and *fread* calls with direct data references. Once the user is finished with the file the call

```
fclose(stream); /* existing call */
```

should be used to deallocate the virtual address space used by the mapped file.

To write map a file the user would:

```
stream = fopen("filename", "w"); /* existing call */
data = fmap(stream, size);      /* new call */
```

where *size* is used as an initial buffer size; if that size is zero, the implementation will choose a suitable size. Initially, the buffer will be zero-filled. Once *fmap* is called, the user may write into any part of the file with an offset less than *bufsize*. An *fwrite* or *fseek* call with an offset greater than *bufsize* will cause an error. To expand the buffer size, the user may call *fmap* again with a larger size parameter. The calls

```
fflush(stream); /* existing call */
fclose(stream); /* existing call */
```

continue to work as before. Similarly, files opened for append and read/write may be *fmaped*.

Table 6-3 shows the time advantage which can be gained by using *fmap* rather than conventional I/O.

Unmapped vs. Mapped Buffered I/O Performance						
Test program	First time user system elapsed I/O				Second time user system elapsed I/O	
old_read	11.5u	3.1s	0:21	481io	11.5u	3.0s 0:21 482io
map_read	11.2u	2.9s	0:15	480io	11.0u	0.9s 0:12 0io

Table 6-3:

Time to read a 1970176 byte file.
(Mach, 4K file system, MicroVAX II)

old_read performs *open*, *mallocs* buffer, calls *read* for whole file and then reads data by array reference.

map_read uses *fopen* and *fmap* and reads data by array reference.

7. The Effect of Mach Memory Mapping on Performance

File mapping is hardly free². Even when a page is already in physical memory, a page fault must be taken on the first process access to validate the corresponding hardware map entries. Currently such a fault takes approximately 1.0-1.4 milliseconds on a MicroVAX II with a 4K page size. There are also several ways in which mapped files can adversely affect performance:

- If the file to be mapped is smaller than a single page, file mapping will always result in a full page being allocated in physical memory with excess data filled with zeroes.
- Mapped files compete with program text and data for physical memory. In a traditional Unix system, user programs maintain a fixed-size buffer, so the buffer cache limits the amount of memory which can be consumed in accessing a file.

Nevertheless, as the performance of the new standard I/O library points out, useful performance gains can be achieved using Mach memory mapping. In fact, because the Mach kernel uses mapped files internally to implement *exec*, overall performance of vanilla 4.3 BSD programs is often improved when run on Mach. Particularly dramatic performance gains are seen on machines where the processor speed is high, memory is plentiful and disk is a bottleneck. For example, performance gains of over 20% have been achieved on a VAX 8650. Improvement can be found, however, even on small memory systems with moderately heavy loads. The multiuser benchmark load used to study the performance of the CMU ITC VICE/VIRTUE file system [6] ran 10-15% faster under Mach than a comparable BSD derived kernel on an IBM RT PC with 4 megabytes of memory.

These performance improvements are especially surprising because many basic operating system overheads are actually larger in Mach than in 4.3 BSD. The use of special purpose scheduling instructions has, for example, been eliminated in the VAX version of Mach. The Mach equivalent of the Unix u-area is not at a fixed address so as to allow multiple threads of control. This increases the cost of task and thread data structure references significantly [9]. In addition, VAX Mach is run as a multiprocessor system even on uniprocessors at CMU, so virtually all kernel operations have had their costs increased by locking concerns.

8. Conclusion

Mach's basic memory primitives provide applications with several mechanisms for sharing memory. As such, a mapped file interface under Mach is not required for shared memory. Mach does provide a non-shared interface for mapped files. This interface is not only appropriate for implementing various applications (e.g. shared libraries and program loading), but has increased both the performance and functionality of the system.

The internal implementation of Mach VM does not preclude shared read/write file mapping. Mach does, in fact, support the 4.2 *mmap* call for the purposes of mapping special device memory

²Unless the output is going to /dev/null!

(typically used for frame buffers). The *mmap* call will also work on normal files but will not map files shared between processes. This restriction was not based on technical issues, but was an intentional modification of the *mmap* semantics. The Mach designers felt it was important to discourage programmers from writing programs which depended on sharing data which might or might not be consistently maintained in a loosely coupled environment.

9. Acknowledgements

The Mach VM implementation was done primarily by Avie Tevanian, Michael Young and David Golub. The implementation has been ported five different machine types and has yet to need modification to accommodate new architectures.

The authors would also like to acknowledge others who have contributed to the Mach kernel including Mike Accetta, Robert Baron, David Black, Jonathan Chew, Eric Cooper, Dan Julin, Glenn Marcy and Robert Sansom. Bob Beck (of Sequent) and Charlie Hill (of North American Philips) helped with the port to the Balance 21000. Fred Olivera and Jim Van Sciver (both formerly of Encore) helped with the port to the MultiMax.

References

- [1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.
- [2] Sequent Computer Systems, Inc.
Dynix Programmer's Manual
Sequent Computer Systems, Inc., 1986.
- [3] Encore Computer Corporation.
UMAX 4.2 Programmer's Reference Manual
Encore Computer Corporation, 1986.
- [4] Leach, P.L., P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf.
The Architecture of an Integrated Local Network.
IEEE Journal on Selected Areas in Communications SAC-1(5):842-857, November, 1983.
- [5] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
Technical Report , Carnegie-Mellon University, February, 1987.
- [6] Satyanarayanan, M., et.al.
The ITC Distributed File System: Principles and Design.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 35-50. ACM, December, 1985.
- [7] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch.
Distributed Transactions for Reliable Systems.
In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985.
Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.
- [8] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D. and Baron, R.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
Technical Report , Carnegie-Mellon University, February, 1987.
- [9] Tevanian, A., Rashid, R., Golub, D., Black, D., Cooper, E., and Young, M.
Mach Threads and the UNIX kernel: The Battle for Control.
Technical Report , Carnegie-Mellon University, April, 1987.

A Replacement for Berkeley Memory Management

Pervaze Akhtar
Gould CSD, Urbana
1101 E. University Ave
Urbana, IL 61801
(217) 384-8597

1. Introduction

Gould's new computer (the NP1) is a symmetrical multi-processor machine with shared memory. The machine is expandable to an 8 cpu configuration. Gould has ported Berkeley 4.3 with substantial modifications to support features such as:

- Suitable for large physical memory systems (NP1 minimum is 64 Mb).
- Able to run without swap space.
- Swap space bound to pages dynamically.
- Pageouts deferred until necessary.
- Supports shared memory - which provides flexibility in use of the address space.
- Copy-on-write supported - the fork and vfork system calls are implemented as a single fork primitive, that utilizes copy-on-write.
- Page replacement is by means of a working set algorithm.
- Supports 1 - 8 processors sharing memory.

4.3 BSD is a single processor implementation of Unix. In order to fully utilize a multi-processor machine, it was clear that significant work would have to be done on the kernel. This paper describes the design and implementation of a replacement memory management system for 4.3 BSD. The reader is assumed to be familiar with the VAX version of 4.3 BSD's Unix, however a concise description of the VAX version of Unix's memory management system may be found in [Mankovich and Kolstad].

2. NP1 Architecture

The Gould NP1 is a 32 bit machine with a 4 Gigabyte virtual and physical address space. The virtual address is broken down as shown in Fig. 1. The hardware has 3 levels of mapping tables consisting of quadrant pointers, segment tables, and page tables. The translation tables reside in physical memory, but for ease of manipulation by Unix, they are mapped into the kernel. Protection is enforced on a per page basis, with 4 independant protection domains being available. The hardware provides a referenced and modified bit (in the

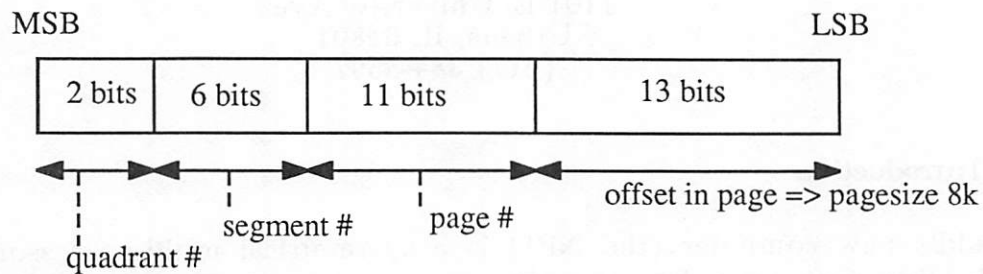


Fig. 1. virtual address structure

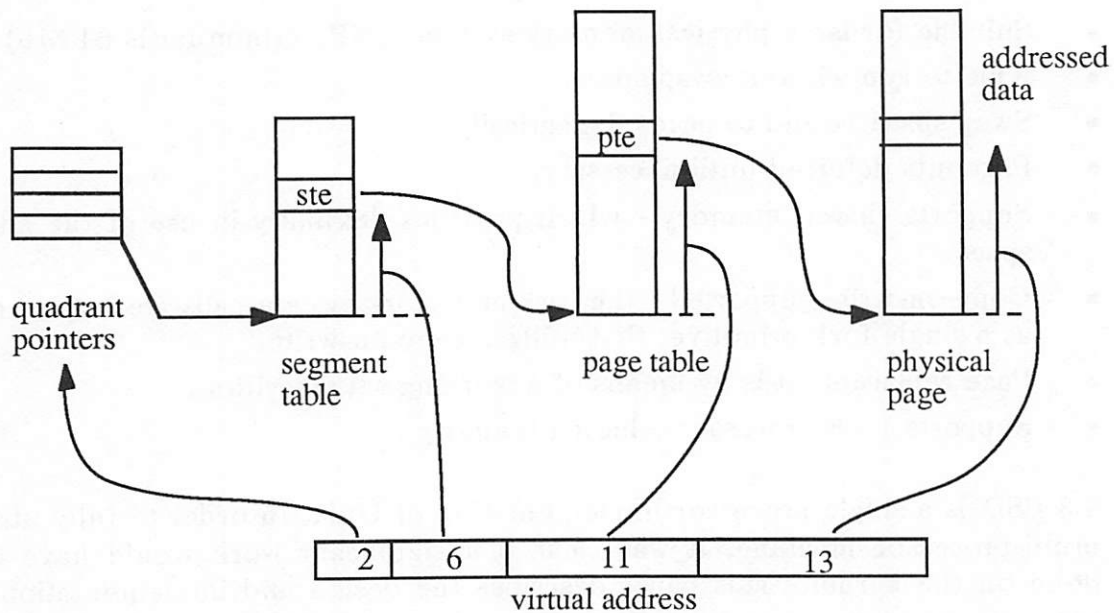


Fig. 2. translation tables

PTE) for each virtual page. The quadrant pointers are the handle that the hardware uses to do virtual to physical address translation. The virtual address space is utilized as shown in Fig. 3. It is split into 2 Gb. of user (quadrants U0 and U1), and 2 Gb of system space (quadrants S0 and S1). The hardware enforces a "fence" between system and user space, generating a trap if an unintentional memory access is made across the "fence". Each processor maintains an internal translation buffer (TB) for optimizing virtual to physical address translation. The translation buffer must be invalidated when a page has its valid bit turned off, or its access protection changed. The instructions provided to flush the TB can:

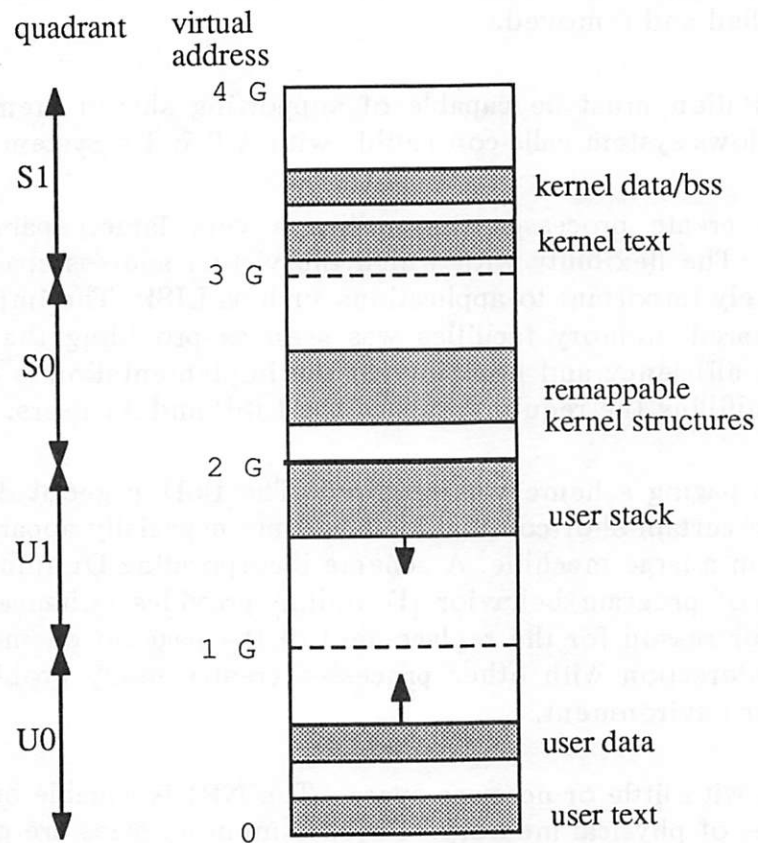


Fig. 3. virtual memory map

- Flush a single page
- Flush all entries for quadrants U0, U1, and S0.
- Flush everything.

Hence the TB management of S0 is linked to user space, even though it is a part of system space. S0 is used for remappable data structures, minimizing the software overhead of TB management.

Flushing affects only the processor that executes it, so translation table changes in memory must be synchronized carefully among the processors.

3. Goals

The required features were:

Concurrency

The memory management system must be capable of use by several CPUs at a time. Many implicit assumptions about exclusive use of data structures must be identified and removed.

Shared memory

The implementation must be capable of supporting shared memory in a manner that allows system calls compatible with AT & T's System V.

Sparse processes

The ability to create processes that utilize a very large, sparse virtual address space. The flexibility with which the virtual address space can be used is extremely important to applications such as LISP. The implementation of the shared memory facilities was seen as providing the required flexibility. The efficiency and flexibility of the implementation is especially important to fulfilling the requirements of the LISP and AI users.

Working Set

A working set paging scheme was required. The BSD pageout daemon is known to have certain shortcomings that become especially apparent when implemented on a large machine. A scheme incorporating Denning's working set model of program behavior [Denning] provides enhanced performance. A major reason for the replacement of the pageout daemon is that its mode of interaction with other processes creates many problems in a multi-processor environment.

No swap space

Be able to run with little or no swap space. The NP1 is capable of addressing 4 Gigabytes of physical memory. Physical memory sizes are growing at rate faster than disk capacity, and it is easy to envisage a machine that is configured with more physical memory than disk storage. BSD Unix's management of swap space is as a severe limitation in this environment.

Modification of the existing memory management system was seen as possible, but providing insufficient return for the effort involved. The modifications would be unlikely to yield a system with the desired flexibility, and would not provide the type of base for future enhancement that was desired.

A port of AT & T's System V (release 3) was studied. It was concluded that the many hardware dependencies and software architecture differences made such a port impractical given the time and resource constraints.

The chosen course of action was to design a memory management system with concepts from System V as a base, with modifications to take maximum advantage of the NP1's requirements and architecture.

4. Key Issues

The key technical and design issues were:

translation buffer

The existence of a per-processor translation buffer, requires that any changes to the memory resident translation tables be carefully synchronized between all processors.

large physical memory

The existence of a large physical memory makes the conventional view of swap space impractical. Unifying the systems view of physical memory and swap space, and removing size assumptions were key design issues.

copy-on-write

The fork system call is expensive because it duplicates all the pages of the calling process. Vfork is efficient, but has functional restrictions. A fork primitive that uses copy-on-write semantics will copy pages only as required, and requires no user level tradeoffs in order to provide optimum performance.

5. Implementation Overview

5.1. Concurrency control

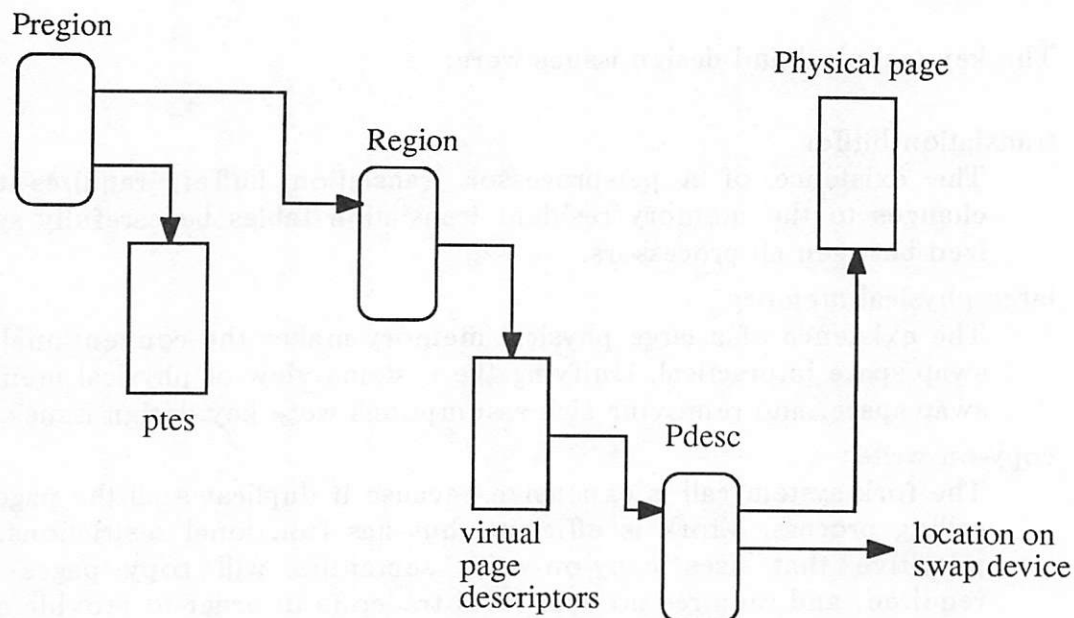
Mutual exclusion on critical data structures and sections is enforced by two types of mutual exclusion primitives, the lock and the critical section. Locks provide for short term locking, where attempts to acquire a locked lock will result in a busy-wait until the resource becomes free. Critical sections (CS) are similar to Dijkstra semaphores in that an attempt to acquire a critical section will result in the process being suspending until the CS becomes free. Both mechanisms are used throughout the kernel for concurrency control.

5.2. Process Structure

Each process consists of a number of logical segments of address space (i.e. text, data, etc). Each logical segment is represented by several levels of data structures.

5.2.1. Regions

All the management of user processes is built around regions. A region describes a block of virtual memory that that is mapped into one or more processes. A particular region may be mapped at different virtual addresses in different processes, but must always start on a 16 Mb boundary. A region may be private to a process (e.g. a stack or data region), or may be shared by many



Process related structures

processes (e.g. text, or shared memory). The region structure contains information such as the file and the offset that it should be loaded from, and the address of a list of virtual page descriptors (VPDs).

5.2.2. Virtual Page Descriptors

A virtual page descriptor consists of either a pointer to a page descriptor structure, or fill-on-demand (FOD) information. Fill-on-demand information signifies a page that has yet to be allocated, and specifies how the contents of that page are to be determined. Fill-on-demand types are:

fill zero

A page is to be allocated and cleared.

no fill

A page is to be allocated, but nothing more. Its current contents are to be overwritten entirely with an I/O transfer or copy operation.

file fill

This page needs to be filled from the file associated with the region. The disk address for the I/O transfer is computed at the time the page is allocated (unlike BSD where the disk address is pre-computed and stored as part of the FOD information).

When a page is actually allocated for some section of address space, the fill-on-demand information is replaced by a pointer to a page descriptor structure.

Page descriptors (pdscs) represent one of the significant changes in low level structures. Page descriptors may be viewed as an extended form of cmap. A pdesc describes where the page is in memory, on disk, or both, and records how many regions and PTEs are referencing the page. The inclusion of a disk address in the pdesc allows the elimination of disk maps. The use of pdescs allows a simple and efficient implementation of copy-on-write, and additional flexibility in memory and swap space reclamation. Pdescs are permanently memory resident (c.f. cmaps) and are maintained in queues according to their states. There are five queues:

Free pdesc

A pool of pdesc structures with no page frame or disk block attached.

Free page

Pdescs with a page frame attached, but no disk block. Not referenced by anyone. This is the primary memory pool used by the memory allocation routines.

Reclaim

Pdescs with a page frame and a disk block. These are pages that have dropped out of a process's working set, but are 'clean'. The reclaim queue is examined by the memory allocators if the free-page queue is exhausted. Page faults on pages in the reclaim queue will result in the appropriate pdesc being removed from the reclaim queue and placed back in the working set of the process.

To Clean

Pdescs with a page frame but no disk block. Pages on this queue have dropped out of the working set of a process but are 'dirty'. Pdescs are taken from this queue, written to disk, and put into the reclaim queue as necessary to free memory.

Sreclaim

"Swap reclaim" queue. Pdescs that are in one or more working sets, and are clean. If the swapper requires swap space because of a shortfall, the swap space associated with pdescs on this queue may be reclaimed.

A pdesc is never on any queue if paging I/O is in progress for it. It will be put in a queue when the I/O completes.

5.2.3. Pregions

Regions are attached to processes by means of pregions. Each process consists of a set of pregions, and each pregon represents one "logical" segment of address space e.g. text, data, stack, etc. The pregon contains information such

as:

- access mode for the region (different processes may wish to access a shared region with different modes).
- Size, the size of the accessible region. A process may attach to a region of the same size or larger. In the latter case, only the size defined in the pre-gion will be accessible to this process.
- PTEs, the page tables required to map in the virtual address space described by the region. The PTEs are constructed with information acquired from the pdescs (attached to the region) and with information from the pre-gion structure.

The information in a pre-gion (including the PTEs) is private to a process. Not sharing PTEs allows an accurate view of page accesses (the working set) to be constructed, without having to compensate for process interactions on shared regions. This multi-level process structure results in some additional overhead and complexity (over System V's implementation of regions). However, one advantage is that PTEs contain no useful information about a process when that process is swapped out. They may be deallocated and returned to the free memory pool resulting in no swap overhead for PTE pages!

5.3. Swap space

The BSD approach to swap space provides a situation where the amount of swap space limits the total number of pages of process virtual space (henceforth referred to as TPVS) that may exist. Our implementation treats swap space as an extension of the physical memory. The sum of the physical memory and swap space determines the TPVS that may be created. Swap blocks are bound to pages dynamically, when page outs are required. In order to prevent deadlock due to TPVS shortfall, swap space accounting is done at the time of an exec/fork/sbrk.

This view of swap space accommodates not only the more traditional situation when secondary storage is some small multiple of physical memory, but also the situation where swap space is small compared to the physical memory, or even a configuration where no swap space is available. [At present our implementation requires a small amount of disk space for an arg map, however it is expected that this limitation will be removed shortly].

5.4. Page-in

The system provides full support for ZMAGIC (demand-loaded executables) and NMAGIC (pre-loaded executables). Due to the protection modes provided on hardware pages, OMAGIC is no longer supported.

5.5. Page-out

The system implemented does not have a pageout daemon like the BSD system. Instead there is a 'working set' scan routine. The working set scan routine is called periodically, in process context, for each process in the system. The scan routine scans the PTEs of a process, and identifies those pages which have not been referenced recently. (i.e. those not in the current working set). These pages are placed in the reclaim queue if they are clean, or the to-clean queue if they are dirty. These operations require a flush of the translation buffer of the processor running this process. If a dirty page has a swap block attached to it, the swap block will be detached from it before it is placed in the to-clean queue. No I/O is scheduled by the scan routine. The working set model does not impose an upper limit on the number of pages a process may have in its working set.

A "page cleaning" daemon process is responsible for examining the to-clean queue and scheduling I/O based upon the systems memory requirements. Disk blocks are allocated and attached to the pdescs at the time the I/O is scheduled.

The paging/swapping I/O system takes advantage of hardware scatter/gather capabilities in order to write a collection of pages onto a single contiguous section of disk, in a single I/O transfer. In order to make this possible, a contiguous set of swap blocks is allocated for each I/O transfer. If the swap space becomes excessively fragmented, the size of the I/O transfers will be tailored to the available segments of contiguous disk space. Reads from the swap device make no attempt to pre-page ('kluster').

The adoption of a working set scheme has 2 advantages:

- Interference - by making page removal a process local activity, interprocess and interprocessor interference is significantly reduced, making the implementation simpler and more reliable.
- Less page faults - the working set model is known to reduce the number of page faults taken by a process during its lifetime [Denning 80] because it more accurately determines the pages required by the process.
- More equitable memory distribution. The size of working set may be arbitrarily large or small based upon the execution characteristics of the process. In an environment with widely varying process sizes, this is a highly desirable characteristic.

5.6. Swapper

The swapper has been modified to provide the necessary synchronization in a multi-processor environment. Some changes were made in the swap in and swap out policies, but the basic mode of operation of the swapper was not

modified.

Since the working set algorithm does not restrict working set sizes in any manner, overcommitment of memory resources is possible. A small overcommitment will cause the pager to remove unreferenced pages. If the pager is unable to free sufficient pages, and a severe memory shortfall develops, the swapper will be activated. The swapper will "remove" processes from memory as required, in order to ensure that the working sets of the remaining processes can be held in memory.

The swapper uses the working set scan routine in order to perform swapouts. A swapout consists of a working set scan, where all pages are forcibly removed from the working set. By using the same mechanism as the paging system, the swapper is able to take advantage of all the I/O optimizations in the paging system. It should also be noted that the swapout of a process does not do any I/O. The swapout will place the pages of the process upon the appropriate pdesc queues. If memory is still needed by the system, the page cleaning daemon will schedule the necessary I/O in order to free pages.

5.7. Future Work

Several further things could be done with the system:

- Swapper - the swapping policy should be examined and modified where necessary. In the situation where paging traffic becomes excessive (thrashing), it would be desirable to suspend the execution of a process, rather than swapping it out. The latter operation may merely serve to drive the paging rate even higher.
- Paging based on I/O traffic - the I/O system keeps various statistics. The pager could use these statistics and try to keep paging traffic balanced across all swap devices.

6. Conclusion

For a relatively small effort, some substantial improvements were made to BSD's memory management system. The changes described provide:

- Expansion capability to larger physical memory sizes.
- Multiprocessor operation.
- Working set model based page management.

This provides a base that can be enhanced and augmented more readily than the original 4.3 BSD Unix.

7. Acknowledgments

David Willcox and Dave Healy (both at Gould CSD Urbana) provided valuable assistance in the design and implementation of the system.

8. References

Babaoglu, Ozalp and Joy, William, "Converting a Swap-Based system to do Paging in an Architecture Lacking Page-Referenced Bits," ACM 1981.

Carr, R.W. and Hennessy, J.L., "WSClock - a Simple and Effective Algorithm for Virtual Memory Management," ACM 1981.

Denning, P., "The Working Set Model for Program Behavior," CACM, Vol. 11, No. 5, May 1968, pp 323-333.

Denning, P., "Working sets past and present," IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, Jan 1980.

Levy, H.M. and Lipman, P.H., "Virtual Memory Management in the VAX/VMS Operating System," IEEE Computer, Mar. 1982, pp 35-42.

Mankovich and Kolstad, "Porting the 4.2 BSD Unix Virtual Memory Subsystem," Proceedings of Winter 1985 Usenix, pp 4-10.

Prieve, B.G and Fabry, R.S., VMIN - An Optimal Variable-Space Page Replacement Algorithm," CACM., Vol. 19, No. 5, May 1976, pp 295-297.

Rodriguez-Rosell, J. and Dupuy, J.P., "The Design, Implementation and Evaluation of a Working Set Dispatcher," CACM., Vol. 16, No. 4. April 1973, pp 247-253.

Virtual Memory Architecture in SunOS

*Robert A. Gingell
Joseph P. Moran
William A. Shannon*

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

A new virtual memory architecture for the Sun implementation of the UNIX[†] operating system is described. Our goals included unifying and simplifying the concepts the system used to manage memory, as well as providing an implementation that fit well with the rest of the system. We discuss an architecture suitable for environments that (potentially) consist of systems of heterogeneous hardware and software architectures. The result is a page-based system in which the fundamental notion is that of mapping process addresses to files.

1. Introduction and Motivation

The UNIX operating system has traditionally provided little support for memory sharing between processes, and no support for facilities such as file mapping. For some communities, the lack of such facilities has been a barrier to the adoption of UNIX, or has hampered the development of applications that might have benefited from their availability. Our own desire to provide a shared libraries capability has provided additional incentive for us to explore providing new memory management facilities in the system.

We have also found ourselves faced with having to support a variety of interfaces. These included the partially implemented interfaces we have had in our 4.2BSD-derived kernel [JOY 83] and those specified by AT&T for System V [AT&T 86]. Aggravating these situations were the variations on those interfaces being developed by a number of vendors that were incompatible with or extended the original proposals. Also, entirely new interfaces have been proposed and implemented, most notably in Carnegie-Mellon's MACH [ACCE 86]. There has been no market movement to suggest which, if any, of these would become dominant, and in some cases a specific interface lacked an important capability (such as System V's lack of file mapping).

Finally, our existing implementation is too constraining a base from which to provide the new functionality we wanted. It is targeted to traditional models of UNIX memory management and specifically towards the hardware model of the VAX.[‡] The work required to enhance the current implementation appeared to be adding its own new wart to an increasingly baroque implementation, and we were concerned for its long-term maintainability.

Thus, we decided to create a new Virtual Memory (VM) system for Sun's implementation of UNIX, SunOS. This paper describes the architecture of this new system: the goals we had for its design and the constraints under which we operated, the concepts it embodies, the interfaces it offers the UNIX application programmer and its relationship to the rest of the system. Although our primary intent is to discuss the architectural issues, information relating to the project and its implementation is provided to add context to the presentation.

[†] UNIX is a trademark of AT&T.

[‡] VAX is a trademark of Digital Equipment Corporation

2. Goals/Non-Goals

Beyond the previously mentioned functional issues of memory sharing and file mapping, our goals for the new architecture were:

- **Unify memory handling.** Our primary architectural goal was to find the general concepts underlying all of the functions we wanted to provide or could envision, and then to provide them as the basis for all VM operations. If successful, we should be able to reimplement existing kernel functions (such as *fork* and *exec*) in terms of these new mechanisms. We also hoped to replace many of the existing memory management schemes in the kernel with facilities provided by the new VM system.
- **Non-kernel implementation of many functions.** If we were successful in identifying and providing the right mechanisms as kernel operations, then it seemed likely that many functions that otherwise would have had to be provided in the kernel could in fact be implemented as library routines. In particular, we wanted to be able to provide capabilities such as shared libraries and the System V interfaces as *applications* of these basic mechanisms.
- **Improved portability.** The existing system was targeted towards a specific machine architecture. In many cases, attributes of this architecture had crept cancerously through the code that implements software-defined functionality. We therefore wanted to describe software-defined objects using data structures appropriate to the software, and relegate machine-dependent code to a lower system layer accessed through a well-defined and narrow interface.
- **Consistent with environment.** We wanted our system to fit well with the UNIX concepts we were not changing. It would not be acceptable to build the world's most wonderful memory management system if it was completely incompatible with the rest of the system and its environment. Particularly important to us in this respect was the use of the file system as the name space for the objects supported by the system. Moreover, we sell systems that are intended to operate in highly networked environments, and thus we could not create a system that presented barriers to the networked environment.

In addition to these *architectural* goals, there were other goals we had for the project as a whole. These *project* goals were:

- **Maintain performance.** Although it is always desirable to tag a project with the label "improves performance", we chose the apparently more conservative goal of simply providing more functionality for the same cost in terms of overall system performance. While the new functionality might enable increased *application* performance, the performance of the system itself seemed uncertain. Further, when one considers that we replaced a mature implementation with one which has not been subjected to several years of tuning, getting back to current performance levels appeared to be an ambitious goal, something later experience has proven correct.
- **Engineer for the future.** We wanted to build an implementation that would be amenable to anticipated future requirements, such as kernel support for "lightweight" processes [KEPE 85] and multiprocessors.

When engaging in a large project, it is often as important to know what one's goals are *not*. In the architectural arena, our principal "non-goals" were:

- **New external interfaces.** As previously noted, a large number of groups were already working on the refinement and definition of interfaces. To the extent possible, we wanted to use such interfaces as had already been defined by others, and to provide those that were sufficiently defined to be implementable and that the market was demanding.
- **Compatible internal interfaces.** An unfortunate characteristic of UNIX is the existence of programs that have some understanding of the system's internals and use this information to rummage through the kernel by reading the memory device. The changes to the system we contemplated clearly made it impossible for us to try to support these programs, and thus we decided not to fool ourselves into trying.

Relevant project non-goals included:

- **Pageable kernel.** We did not intend to produce an implementation in which the kernel itself was paged – beyond a general desire in principle for the kernel to use less physical memory, we would have satisfied no specific functional goal by having the kernel pageable. However, it has turned out that a considerable portion of the memory that was previously “wired down” for kernel use is in fact now paged, although kernel code remains physically locked.
- **Massive policy changes.** Our interests lay in changing the mechanisms and what they provide, not in the policies by which they were administered. Although we would eventually like to support an integrated view of process and memory scheduling using techniques such as working set page replacement policies and balance set scheduling, we decided to defer these to future efforts.

3. Constraints

Working within the framework of an existing system imposed a number of constraints on what we could do. The constraints were not always limits on our flexibility; in fact, those reflecting specific customer requirements provided data that guided us through a number of design decisions. A major constraint was that of compatibility with previous versions of the system – ultimately, compatibility drove many decisions.

One such decision was that the new system would execute existing *a.out* files. This was necessary to preserve the utility of the programs already in use by customers and third parties. An important implication is that the system must provide a binary-compatible interface for existing programs, which means that existing system calls that perform memory management functions must continue to work. In our case, this meant supporting our partial implementation of the 4.2BSD *mmap*(2) system call, which we used to map devices, such as frame buffers, into a process’s address space.

Although the system had to be binary-compatible, we did not feel constrained to leave it source-compatible, nor to use *mmap* as the principal interface to the memory management facilities of the system. Users with programs that used interfaces we changed in this manner would have to change their programs the next time they compiled them, but they would not be forced to recompile just to install and continue operating on the new system.

A wide variety of customer requirements implied that the interfaces we would offer would have to present very few constraints on a process’s use of its address space. Some applications wanted to manage their address space completely, including the ability to assign process addresses for objects and to use a large, sparsely populated address space. Our own desire to build a base on which many different interfaces could be easily constructed suggested that we wanted as much flexibility as possible in user level address space management. However, other factors and requirements suggested that the system should also be able to control many details of an address space. One such factor was the introduction of a virtual address cache in the Sun-3/200 family of processors, where system control of address assignment would have a beneficial impact on performance. We also wanted to use copy-on-write techniques to enhance the level of sharing in the system, and to do this efficiently required page-level protection.

4. New Architecture: General Concepts

This section describes in general terms the abstractions and properties of the new VM system, and some reflections on the decisions that led to their creation. In many cases, our decisions were not based on obvious considerations, but rather “fell out” of a large number of small issues. Although this makes the decisions more difficult to explain, the process by which they were reached increased our confidence that, given our goals and constraints, we had in fact reached the best conclusion.

4.1. Pages vs. Segments

Our earliest decision was that the basic kernel facilities would operate on pages, rather than segments. The major factors in this decision included:

- compatibility with current systems (the 4.2BSD *mmap* is page-based);

- implementing efficient copy-on-write facilities required maintenance of per-page information anyway;
- pages appeared to offer the greatest opportunity to satisfy customer requirements for flexibility; and
- segments could be built as an abstraction on top of the page-based interface by library routines.

The major advantage to a segment-based mechanism appeared simply to be that it was a “better” programming abstraction. Since we could still build abstraction from the page-based mechanisms, and in fact gained some flexibility in building different forms of the abstraction as libraries, providing segments through the kernel appeared to offer little benefit and possibly even presented barriers to accomplishing some of our goals.

Although we believed we could gain the architectural advantages of segments through library routines built on our page-based system, another potential advantage to a segment-based system was the opportunity to implement a compact representation for a sparsely populated address space. However, since we needed per-page information to implement per-page copy-on-write and perform other physical storage management, at the very least we would end up with a mix of page- and segment-oriented data structures. We recognized that we could keep the major implementation advantage of a segment-based system, i.e., the concise description of the mapping for a range of addresses, by viewing it as an optimization (a sort of run-length encoding) of the per-page data structure (a similar scheme is used in MACH.)

4.2. Virtual Memory, Address Spaces, and Mapping

The system’s *virtual memory* consists of all its available physical memory resources. Examples include file systems (both local and remote), pools of unnamed memory (also known as *private* or *anonymous* storage, and implemented by the processor’s primary memory and *swap space*), and other random access memory devices. Named objects in the virtual memory are referenced through the UNIX file system. This does not imply that all file system objects are in the virtual memory, but simply that all named objects in the virtual memory are named in the file system. One of the strengths of UNIX has been the use of a single name-space for system objects, and we wished to build upon that strength. Some objects in the virtual memory, such as process private memory and our implementation of System V shared memory segments, do not have names. Although the most common form of object is the UNIX “regular file”, previous work on SunOS has allowed for many different implementations of objects, which the system manipulates as an abstraction of the original UNIX *inode*, called a *vnode* [KLEI 86].

A process’s *address space* is defined by mappings onto the address spaces of one or more objects in the system’s virtual memory. As previously discussed, the system provides a page-based interface, and thus each mapping is constrained to be sized and aligned with the page boundaries defined by the system on which the process is executing. Each page may be mapped (or not) independently, and thus the programmer may treat an address space as a simple vector of pages. It should be noted that the only valid process address is one which is mapped to some object, and in particular there is no memory associated with the process itself – all memory is represented by virtual memory objects.

Each object in the virtual memory has an *object address space* defined by some physical storage, the specific form being object-specific. A reference to an object address accesses the physical storage that implements the address within the object. The virtual memory’s associated physical storage is thus accessed by transforming process addresses to object addresses, and then to the physical store. The system’s VM management facilities may interpose one or more layers of logical caching on top of the actual physical storage used to implement an object, a fact that has implications for *coherency*, discussed below.

A given process page may map to only one object, although a given object address may be the subject of many process mappings. The amount of the object’s address space covered by a mapping is an integral multiple of the page size as seen by the process performing the mapping. An important characteristic of a mapping is that the object to which the mapping is made is not required to be affected by the mere *existence* of the mapping. The implications of this are that it cannot, in general, be expected that an object has an “awareness” of having been mapped, or of which portions of its address space are accessed by mappings; in particular, the notion of a “page” is not a property of the object. Establishing a mapping

to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable by a process. Applications may find it advantageous to access the storage resources they use directly rather than indirectly through *read* and *write*. Potential advantages include efficiency (elimination of unnecessary data copying) and reduced complexity (e.g., updates changed to a single step rather than a *read*, modify buffer, *write* cycle). The ability to access an object and have it retain its identity over the course of the access is unique to this access method, and facilitates the sharing of common code and data.

It is important to note that this *access method* view of the VM system does not directly provide sharing. Thus, although our motivations included providing shared memory, we have actually only provided the mechanisms for applications to *build* such sharing. For the system to provide not only an access method but also the *semantics* for such access is not only difficult or impossible, it is not clear that it is the correct thing to do in a highly heterogeneous environment. However, useful forms of sharing can be built in such environments, as the previous mechanisms for sharing in the kernel (such as the shared program text and file data buffer cache) have been subsumed by kernel programming building on top of these mechanisms.

4.3. Networking, Heterogeneity, and Coherence

Many of the factors that drove our adoption of the access method view of a VM system originated from our goal of providing facilities that “fit” with their expected environment. A major characteristic of our environment is the extensive use of networking to access file systems that would be part of the system's virtual memory. These networks are not constrained to consist of similar hardware or a common operating system; in fact, the opposite is encouraged. Making extensive assumptions about the properties of objects or their access creates potentially extensive barriers to accommodating heterogeneity. These properties include such system variables as page sizes and the ability of an object to synchronize its uses. While a given set of processes may *apply* a set of mechanisms to establish and maintain various properties of objects, a given operating system should not *impose* them on the rest of the network.

As it stands, the access method view of a virtual memory maintains the potential for a given object (say a text file) to be mapped by systems running our memory management system but also accessed by systems for which the notion of a virtual memory or storage management techniques such as paging would be totally foreign, such as PC-DOS. Such systems could continue to share access to the object, each using and providing its programs with the access method appropriate to that system. The alternative would be to prohibit access to the object by less capable systems, an alternative we find unacceptable.

A new consideration arises when applications use an object as a communications channel, or otherwise attempt to access it simultaneously. In addition to providing the mapping functions described previously, the VM management facilities also manage a storage hierarchy in which the processor's primary memory is often used as a cache for data from the virtual memory. Since the system cannot assume either that the object will coordinate accesses to it, nor that other systems will in fact cooperate with such coordination, it does not attempt on its own to synchronize the “virtual memory cache” it maintains. This is not to say that such objects can not exist, nor that systems will not cooperate; simply that *in general* the system can not make such an assumption. Even within a single system, the sharing that results is a consequence of the system's attempt to use its cache resources efficiently, not part of its defined functionality.

However, the lack of cache synchronization is not the limitation it might first appear. Applications that intend to share an object must employ a synchronization mechanism around their access and this requirement is independent of the access method they use. The scope and nature of the mechanism employed is best left to the application to decide. While today applications sharing a file object must access and update it indirectly using *read* and *write*, they must coordinate their access using semaphores or file locking or some application-specific protocol. In such environments, either caching is totally disabled (resulting in performance limitations) or the applications must employ a function such as *fsync* to ensure that the object is updated. Coherency of shared objects is not a new issue, and the introduction of a new access method simply exposes a new manifestation of an old problem. All that is required in an environment where mapping replaces *read* and *write* as the access method is that an operation comparable to *fsync* be provided.

Thus, the nature and scope of synchronization over shared objects is something that is application-defined from the outset. If the system attempted to impose any automatic semantics for sharing, it might prohibit other useful forms of mapped access that have nothing whatsoever to do with communication or sharing. By providing the mechanism to support coherency, and leaving it to cooperating applications to apply the mechanism, our design meets the needs of applications without providing barriers to heterogeneity. Note that this design does not prohibit the creation of libraries that provide coherent abstractions for common application needs. Not all abstractions on which an application builds need be supplied by the “operating system”.

4.4. Historical Acknowledgements

Many of the concepts we have described are not new. MULTICS [ORGA 72] supported the notion of file/process memory integration that is fundamental to our system. TENEX [BOBR 72] [MURP 72] supported a page-based environment together with the notion of a process page map independent of the object being mapped.

5. External Interfaces: System Calls

The applications programmer gains access to the facilities of the new VM system through several sets of system calls. At present, we have defined our principal interface to be a refinement of those provided with 4.2BSD. We also provide interfaces for System V's shared memory operations. The new system also impacted other system calls and facilities. These are described further below. Although these represent the initial interfaces we intend to support, others may be provided in the future in response to market demand.

5.1. 4.2BSD-based Interfaces

The 4.2BSD UNIX specification [JOY 83] included the definition of a number of system calls for mapping files, although the system did not implement them. Earlier releases of SunOS included partial implementations of these calls to support mapping devices such as frame buffers into a process's address space. The basic concepts embedded in the interface were very close to our own, namely a page-based system providing mappings from process addresses to objects identified with file descriptors, and thus working from this base was a natural thing to do.

However, we had problems with the 4.2BSD interfaces due to their sketchy definition. Although the intent was well understood, the lack of an implementation left many semantic issues unresolved or ambiguous. We required some facilities that were not part of the specification, and other facilities were part of the specification but seemed superfluous. Thus, although we did manage to avoid creating an entirely new interface, we did find ourselves refining an existing, but unimplemented one. The process of refinement involved many people; in fact most were external to Sun and involved exchanges utilizing a “VM interest” mailing list supported and maintained by the developers at UC Berkeley, CSRG. Table 1 summarizes our refined interface, and the following sections expand on various areas of refinements.

5.1.1. *mmap*

The *mmap*(2) system call is used to establish mappings from a process's address space to an object. Its definition is:

caddr_t mmap(addr, len, prot, flags, fd, off)

mmap establishes a mapping between the process's address space at an address *paddr* for *len* bytes to the object specified by *fd* at offset *off* for *len* bytes. The value of *paddr* is an implementation-dependent function of the parameter *addr* and values of *flags*, further described below. A successful *mmap* call returns *paddr* as its result. The address ranges covered by [*paddr*, *paddr* + *len*) and [*off*, *off* + *len*) must be legitimate for the address space of a process and the object in question, respectively. The mapping established by *mmap* replaces any previous mappings for the process's pages in the range [*paddr*, *paddr* + *len*).

The parameter *prot* determines whether *read*, *execute*, *write* or some combination of accesses are permitted to the pages being mapped. The values desired are expressed by or'ing the flags values PROT_READ, PROT_EXECUTE, and PROT_WRITE. It is not expected that all implementations

Table 1 – Refined 4.2BSD Interfaces	
Call	Function
<code>madvise(addr, len, behav)</code> <code>caddr_t addr; int len, behav;</code>	Gives advice about the handling of memory over a range of addresses.
<code>mincore(addr, len, vec)</code> <code>caddr_t addr; int len; result char *vec;</code>	Determines residency of memory pages. (Will be replaced by more general map reading function.)
<code>caddr_t</code> <code>mmap(addr, len, prot, flags, fd, off)</code> <code>caddr_t addr; int len, prot, flags, fd;</code> <code>off_t off;</code>	Establish mapping from address space to object named by <code>fd</code> .
<code>mprotect(addr, len, prot)</code> <code>caddr_t addr; int len, prot;</code>	Change protection on mapped pages.
<code>msync(addr, len, flags)</code> <code>caddr_t addr; int len, flags;</code>	Synchronizes and/or invalidates cache of mapped data.
<code>munmap(addr, len)</code> <code>caddr_t addr; int len;</code>	Removes mapping of address range.

literally provide all possible combinations. `PROT_WRITE` is often implemented as `PROT_READ|PROT_WRITE`, and `PROT_EXECUTE` as `PROT_READ|PROT_EXECUTE`. However, no implementation will permit a write to succeed where `PROT_WRITE` has not been set. The behavior of `PROT_WRITE` can be influenced by setting `MAP_PRIVATE` in the *flags* parameter.

The parameter *flags* provides other information about the handling of the pages being mapped. The options are defined by a field describing an enumeration of the “type” of the mapping, and a bit-field specifying other options. The enumeration currently defines two values, `MAP_SHARED` and `MAP_PRIVATE`. The bit-field values are `MAP_FIXED` and `MAP_RENAME`. The “type” value chosen determines whether stores to the mapped addresses are actually propagated to the object being mapped (`MAP_SHARED`) or directed to a copy of the object (`MAP_PRIVATE`). If the latter is specified, the initial write reference to a page will create a private copy of the page of the object and redirect the mapping to the copy. The mapping type is retained across a `fork(2)`. The mapping “type” only affects the disposition of stores by *this* process – there is no insulation from changes made by other processes. If an application desires such insulation, it should use the *read* system call to make a copy of the data it wishes to keep protected.

`MAP_FIXED` informs the system that the value of *paddr* must be *addr*, exactly. The use of `MAP_FIXED` is discouraged, as it may prevent an implementation from making the most effective use of system resources.

When `MAP_FIXED` is not set, the system uses *addr* as a hint in an implementation-defined manner to arrive at *paddr*. The *paddr* so chosen will be an area of the address space that the system deems suitable for a mapping of *len* bytes to the specified object. All implementations interpret an *addr* value of zero as granting the system complete freedom in selecting *paddr*, subject to constraints described below. A non-zero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for *paddr*, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack “segments”. In the current SunOS implementation, the system strives to choose alignments for mappings that maximize the performance of systems with a virtual address cache.

`MAP_RENAME` causes the pages currently mapped in the range [*paddr*, *paddr* + *len*) to be effectively renamed to be the object addresses in the range [*off*, *off* + *len*). The currently mapped pages must be mapped as `MAP_PRIVATE`. `MAP_RENAME` implies a `MAP_FIXED` interpretation of *addr*. *fd* must be open for write. `MAP_RENAME` affects the size of the memory object referenced by *fd*: the size is $\max(\text{off} + \text{len} - 1, \text{flen})$ (where *flen* was the previous length of the object). After the pages are renamed, a mapping

to them is reestablished with the parameters as specified in the renaming *mmap*.

The addition of `MAP_FIXED` and corresponding changes in the default interpretation of *addr* and *mmap*'s return value represent the principal change made to the original 4.2BSD specification. The change was made to remove the burden of managing a process's address space from applications that did not wish it.

5.1.2. Additions

We added one new system call, *msync*. *msync* has the interface

`msync(addr, len, flags)`

msync causes all modified copies of pages over the range [*addr*, *addr* + *len*) in system caches to be flushed to the objects mapped by those addresses. *msync* optionally invalidates such cache entries so that further references to the pages will cause the system to obtain them from their permanent storage locations. The *flags* argument provides a bit-field of values which influences *msync*'s behavior. The bit names and their interpretations are:

<code>MS_ASYNC</code>	Return immediately
<code>MS_INVALIDATE</code>	Invalidate caches

`MS_ASYNC` causes *msync* to return immediately once all I/O operations are scheduled; normally, *msync* will not return until all I/O operations are complete. `MS_INVALIDATE` causes all cached copies of data from mapped objects to be invalidated, requiring them to be re-obtained from the object's storage upon the next reference.

5.1.3. Unchanged Interfaces

Two 4.2BSD calls were implemented without change. They were *mprotect* for changing the protection values of mapped pages, and *munmap* for removing a mapping.

5.1.4. Removed: *mremap*

We deleted one system call, *mremap*. Upon reading the 4.2BSD specification, we had the impression that *mremap* was the mapping equivalent of the UNIX *mv* command. However, discussions with those involved in its original specification created confusion as to whether it was in fact supposed to be the equivalent of *mv*, *cp*, or *ln*. In the presence of the uncertainty and lacking any other motivation to include it, *mremap* was dropped from the system.

5.1.5. Open Issues

Two 4.2BSD system calls, *madvise* and *mincore*, remain unspecified. *madvise* is intended to provide information to the system to influence its management policies. Since a major rework of such policies was deferred to a future release, we decided to defer full specification and implementation of *madvise* until that time.

mincore was specified to return the residency status of a group of pages. Although the intent was clear, we felt that a more comprehensive interface for obtaining the status of a mapping was required. However, at present, this revised interface has not been defined.

Also unspecified is an interface for locking pages in memory. We envision either a new *mlock* system call, or a variation on *madvise*.

5.2. System V Shared Memory

The "System V Interface Definition" [AT&T 86] defines a number of operations on entities called "shared memory segments". Early in our project, we had hoped to implement these operations not as system calls but rather as library routines which built the System V abstractions out of the basic mechanisms supplied by the kernel. Unfortunately, System V shared memory is almost, but not completely the same as, a UNIX file. The primary differences are:

- **name space:** a shared memory segment exists in a name space different from that of the traditional UNIX file system; and
- **ownership and access:** a shared memory segment separates the notion of “creator” from “owner”.

Together, these differences motivated a kernel-based implementation to allocate and manage the different name space (which shared implementation with other System V-specific objects such as semaphores), and to administer the different ownership and access control operations.

Although the databases peculiar to these differences are maintained inside the kernel, the implementation of the objects and access are built from the standard notions. Specifically, the memory object representing the shared memory segment exists as an unnamed object in the system’s virtual memory, and the operation which attaches processes to it performs the internal equivalent of an *mmap*.

Implementation plans call for the object used to represent the shared memory segment to be supported by an anonymous memory-based file system. */tmp* could be implemented as a file system of this type, potentially eliminating all I/O operations for temporary files and simply supporting them out of the processor’s memory resources.

5.3. Other System Calls and Facilities

The new VM system has had an impact on other areas of the system as well, either extending or slightly altering the semantics of existing operations.

5.3.1. “Segments”

Traditionally, the address space of a UNIX process has consisted of three segments: one each for write-protected program code (text), a heap of dynamically allocated storage (data), and the process’s stack. Under the new system, a process’s address space is simply a vector of pages and there exists no real structure to the address space. However, for compatibility purposes, the system maintains address ranges that “should” belong to such segments to support operations such as extending or contracting the data segment’s “break”. These are initialized when a program is initiated with *exec*.

5.3.2. *exec*

exec overlays a process’s address space with a new program to be executed. Under the new system, *exec* performs this operation by performing the internal equivalent of an *mmap* to the file containing the program. The text and initialized data segments are mapped to the file, and the program’s uninitialized data and stack areas are mapped to unnamed objects in the system’s virtual memory. The boundaries of the mappings it establishes are recorded as representing the traditional “segments” of a UNIX process’s address space.

exec establishes MAP_PRIVATE mappings, which has implications for the operation of *fork* and *ptrace*, as discussed below. The text segment is mapped with only PROT_READ and PROT_EXECUTE protections, so that write references to the text produce segmentation violations. The data segment is mapped as writable; however any page of initialized data that does not get written may be shared among all the processes running the program.

5.3.3. *fork*

Previously, a process created by *fork* had an address space made from a copy of its parent’s address space. Under the new system, the address space is not copied, but the mappings defining it are. Since *exec* specifies MAP_PRIVATE on all the mappings it performs, parent and child thus effectively have copy-on-write access to a single set of objects. Further, since the mapping is generally far smaller than the data it describes, *fork* should be considerably more efficient. Any MAP_SHARED mappings in the parent are also MAP_SHARED in the child, providing the opportunity for both parent and child to operate on a common object.

5.3.4. *vfork*

Berkeley-based systems include a “VM-efficient” form of the *fork* system call to avoid the overhead of copying massive processes that simply threw away the copy operation with a subsequent *exec* call. At one point we hoped that the efficiencies gained through a reimplemented *fork* would obviate the need for *vfork*. Unfortunately, *vfork* is defined to suspend the parent process until the child performs either an *exec* or an *exit* and to allow the child full access to the parent’s address space (*not* a copy) in the interim. A number of programs take advantage of this quirk, allowing the child to record data in the address space for later examination by the parent. Eliminating *vfork* would break these programs, a fact we discovered in numerous ways when early versions of the system simply treated a *vfork* as *fork*. Further, *vfork* remains fundamentally more efficient than even a *fork* that only copies an address space map, since *vfork* copies nothing.

However, to encourage programmers at Sun to avoid the use of *vfork*, we took our time restoring it to the system and as a result got many programs “fixed”.

5.3.5. *ptrace*

In previous versions of the system, the *ptrace* system call (used for process debugging) would refuse to deposit a breakpoint in a program that was being run by more than one process. This restriction was imposed by the nature of the old system’s facility for sharing program code, which was to share the entire text portion of an executable file.

In the new system, the system simply shares file pages among all those who have mappings to them. When a mapping is made MAP_PRIVATE, writes by a process to a page to which writes are permitted are diverted to a copy of the page – leaving the original object unaffected. *ptrace* takes advantage of the fact that an *exec* establishes the mapping to the file containing the program and its initialized data as MAP_PRIVATE, as it inserts a breakpoint by making a read-only page writable, depositing the breakpoint, and restoring the protection. The page on which the breakpoint is deposited, and only that page, is no longer shared with other users of the program – and their view of that page is unaffected.

5.3.6. *truncate*

The *truncate* system call has been changed so that it sets the length of a file. If the newly specified length is shorter than the file’s current length, *truncate* behaves as before. However, if the new length is longer, the file’s size is increased to the desired length. When writing a file exclusively through mapping, extending through *truncate* is the only alternative to MAP_RENAME operations for growing a file.

5.3.7. Resource Limits

Berkeley-based systems include functions for limiting the consumption of certain system resources. We have introduced a new resource limit: RLIMIT_PRIVATE. This limit controls the amount of “private memory” that a process may dynamically allocate from the system’s source of unnamed backing store. In many respects, RLIMIT_PRIVATE really describes the limit that RLIMIT_DATA and RLIMIT_STACK attempt to capture, namely the amount of swap space a given process may consume.

6. Internal Interfaces

The new VM system provides a set of abstractions and operations to the rest of the kernel. In many cases, these are used directly as the basis for the system call interfaces described above. In other areas they support internal forms of those system call interfaces, allowing the kernel to perform mappings for the address space in which it operates. The VM system also relies on services from other areas of the kernel.

6.1. Internal Role of VM

In general, the kernel uses the VM system as the manager of a logical cache of memory pages and as the object manager for “address space objects”. In its role as cache manager, the VM system also manages the physical storage resources of the processor, as it uses these resources to implement the cache it maintains. The VM system is a particularly effective cache manager, and maintains a high degree of sharing over multiple uses of a given page of an object. As such, it has subsumed the functions of older

data structures, in particular the text table and disk block data buffer cache (the “buffer cache”). The VM system has replaced the old fixed-size buffer cache with a logical cache that uses all of the system’s pageable physical memory. Thus its use as a “buffer cache” in the old sense dynamically adapts to the pattern of the system’s use — in particular if the system is performing a high percentage of file references, all of the system’s pageable physical memory is devoted to a function that previously only had approximately 10% of the same resources. The VM system is also responsible for the management of the system’s memory management hardware, although these operations are invisible to the machine-independent portions of the kernel.

Kernel algorithms that operate on logical quantities of memory, such as the contents of file pages, do so by establishing mappings from the kernel’s address space to the object they wish to access. Those algorithms that implement the *read* and *write* system calls on such memory objects are particularly interesting: they operate by creating a mapping to the object and then copying the data to or from user buffers as appropriate. When mapping is used in this manner, users of the object are provided with a consistent view of the object, even if they mix references through mapped accesses or the *read* and *write* system calls. Note that the decision to use mapping operations in this way is left to the manager of the object being accessed.

The VM system does not know the semantics of the UNIX operating system. Instead, those properties of an address space that are the province of UNIX, such as the notions of “segments” and stack-growth, are implemented by a layer of UNIX semantics over the basic VM system. By providing only the basic abstractions from the VM system itself, we believe we have made it easier to provide future system interfaces that may not have UNIX-like characteristics.

The VM system relies on the rest of the system to provide managers for the objects to which it establishes mappings. These managers are expected to provide advice and assistance to the VM system to ensure efficient system management, and to perform physical I/O operations on the objects they manage. These responsibilities are detailed further below.

6.2. *as* layer

The primary object managed by the VM system is a (process) *address space* (*as*). The interfaces through which the system requests operations on an *as* object are summarized in Table 2, and are collectively referred to as the *as*-layer of the system. An *as* contains the memory of the mappings that comprise an address space. In addition, it contains a *hardware address translation* (*hat*) structure that holds the state of the memory management hardware associated with this address space. This structure is opaque to much of the VM system, and is interpreted only by a machine-dependent layer of the system, described further below.

An *as* exists independent of any of its uses, and may be shared by multiple processes, thus setting the stage for future integration of a multi-threaded address space capability as described in [KEPE 85]. The “address space” in which the kernel operates is also described by an *as* structure, and is the handle by which the kernel effects internal mapping operations using *as_map*.

The operations permitted on an *as* generally correspond to the functions provided by the system call interface. An implication of this is that just about any operation that the kernel could perform on an address space could also be implemented by an application directly. More work is necessary to define an interface for obtaining information about an *as*, to support the generation of *core* files, and the as-yet unspecified interfaces for reading mappings. An additional interface is also needed to support any advice operations we might choose to define in the future.

Internally to an address space, each individual mapping is treated as an object with a “mapping object manager”. Such mappings are run-length compact encodings describing the mapping being performed, and may or may not have per-page information recorded depending on the nature of the mapping or subsequent references to the object being mapped. Due to a regrettable lack of imagination at a critical junction in our design, these “mapping objects” are termed *segments*, and their managers are called “segment drivers”.

Table 2 – <i>as</i> operations	
Operation	Function
<code>struct as *as_alloc()</code>	<i>as</i> allocation.
<code>struct as *as_dup(as)</code> <code>struct as *as;</code>	Duplicates <i>as</i> – used in <i>fork</i> .
<code>void as_free(as)</code> <code>struct as *as;</code>	<i>as</i> deallocation.
<code>enum as_res</code> <code>as_map(as, addr, size, crfp, crargsp)</code> <code>struct as *as; addr_t addr; u_int size;</code> <code>int (*crfp)(); caddr_t crargsp;</code>	Internal <i>mmap</i> . Establish a mapping to an object using the mapping manager routine identified in <i>crfp</i> , providing object specific arguments in the opaque structure <i>crargsp</i> .
<code>enum as_res</code> <code>as_unmap(as, addr, size)</code> <code>struct as *as; addr_t addr; u_int size;</code>	Remove a mapping in <i>as</i> .
<code>enum as_res</code> <code>as_setprot(as, addr, size, prot)</code> <code>struct as *as; addr_t addr;</code> <code>u_int size, prot;</code>	Alter protection of mappings in <i>as</i> .
<code>enum as_res</code> <code>as_checkprot(as, addr, size, prot)</code> <code>struct as *as; addr_t addr;</code> <code>u_int size, prot;</code>	Determine whether mappings satisfy protection required by <i>prot</i> .
<code>enum as_res</code> <code>as_fault(as, addr, size, type, rw)</code> <code>struct as *as; addr_t addr; u_int size;</code> <code>enum fault_type type; enum seg_rw rw;</code>	Resolves a fault.
<code>enum as_res</code> <code>as_faulta(as, addr, size)</code> <code>struct as *as; addr_t addr; u_int size;</code>	Asynchronous fault – used for “fault-ahead”.

6.3. *hat* layer

As previously noted, a *hat* is an object representing an allocation of memory management hardware resources. The set of operations on a *hat* are not visible outside of the VM system, but represent a machine-dependent/independent boundary called the *hat*-layer. Although it provides no services to the rest of the system, the *hat*-layer is of import to those faced with porting the system to various hardware architectures. It provides the mapping from the software data structures of an *as* and its internals to those required by the hardware of the system on which it resides.

We believe that the *hat*-layer has successfully isolated the hardware-specific requirements of Sun's systems from the machine-independent portions of the VM system and the rest of the kernel. In particular, under the old system the addition of support for a virtual address cache permeated many areas of the system. Under the new system, support for the virtual address cache is isolated within the *hat* layer.

6.4. I/O Layer

The primary services the VM system requires of the rest of the kernel are physical I/O operations on the objects it maps. These operations occur across an interface called the “I/O Layer”. Although used mainly to cause physical page frames to be filled (page-in) or drained (page-out) operations, the I/O layer also provides an opportunity for the managers of particular objects to map the system-specific page

abstraction used by the VM system to the representation used by the object being mapped.

For instance, although the system operates on page-sized allocations, the 4.2BSD UNIX file system [MCKU 84] operates on collections of disk blocks that are often not page-sized. Efficient file system performance may also require non-page-sized I/O operations, in order to amortize the overhead of starting operations and to maximize the throughput of the particular subsystem involved. Thus, the VM system will pass several operations (such as the resolution of a fault on an object address, even one for which the VM system has a cached copy) through the object manager to provide it the opportunity to intercede. The object manager for NFS files uses these intercessions to prevent cached pages from becoming stale. Managers for network-coherent objects enforce coherence through this technique.

The I/O layer is to some extent bi-directional, as a given operation requested by the VM system may cause the object manager to request several VM-based operations. I/O clustering is an example of this, where a request by the VM system to obtain a page's worth of data may cause the object manager to actually schedule an I/O operation for logical pages surrounding the one requested in the hopes of avoiding future I/O requests. The old notion of "read-ahead" is implemented in this manner, and each object manager has the opportunity to recognize and act on patterns of access to a given object in a manner that maximizes its performance.

7. Project Status & Future Work

The architecture described in this paper has been implemented and ported to the Sun-2 and Sun-3 families of workstations. At present, all our major functional goals have been met. The work has consumed approximately four man-years of effort over a year and a half of real time. A surprisingly large amount of effort has been drained by efforts to interpose the VM system as the logical cache manager for the file systems, in particular with respect to the 4.2BSD UNIX file system.

With respect to our performance goals, more tuning work is required before we can claim to meet them. However, in some areas dealing with file access, early benchmarks reveal substantial performance improvements resulting from the much larger cache available for I/O operations. We expect further performance improvements when more of the system uses the new mechanisms. In particular, we expect an implementation of shared libraries to have a substantial impact upon the use of system resources. Future uses of mapping include a rewritten standard I/O library to use *mmap* rather than *read* and perhaps *write*, thus eliminating the dual copying of data and providing a transparent performance improvement to many applications. As sharing increases in the system, we expect the requirements for swap resources to decrease.

Other future work involves refining and completing the interfaces that have not yet been fully defined. We plan an investigation of new management policies, especially with respect to different page-replacement policies and the better integration of memory and processor scheduling. We would also like to port the system to different hardware bases, in particular to the VAX, to test the success of the *hat* layer in isolating machine dependencies from the rest of the system.

8. Conclusions

We believe the new VM architecture successfully meets our goals. Reviewing these reveals:

- **Unify memory handling.** All VM operations have been unified around the single notion of file mapping. Extant operations such as *fork* and *exec* have been reconstructed and their performance, and in some cases function, has been improved through their use of the new mechanisms.
- **Non-kernel implementation of many functions.** Although we were disappointed that kernel support was required to implement System V shared memory segments, we believe that this goal has been largely satisfied. In particular, our implementation of shared libraries [GING 87] requires no specific kernel support. We believe the basic operations the interfaces provide will permit the construction of other useful abstractions with user-level programming.
- **Improved portability.** Although more experience is required, we were pleased with the degree to which the Sun-3 virtual address cache was easily incorporated into the new system, in comparison with the difficulty experienced in integrating it into the previous system.

- **Consistent with environment.** The new system builds on the abstractions already in UNIX, in particular with respect to our use of the UNIX file system as the name space for named virtual memory objects. The integrated use of the new facilities in the system has helped to extend the previous abstractions in a natural manner. The semantics offered by the basic system mechanisms also do not impede the heterogeneous use of objects accessed through the system, an important consideration for the networked environments in which we expect the system to operate.

Finally, we have provided the functionality that motivated the work in the first place.

9. Acknowledgements

The system was designed by the authors, with Joe Moran providing the bulk of the implementation. Bill Joy offered commentary and advice on the architecture, as well as insights into the intents of the 4.2BSD interface, and an initial sketch of an implementation of the internal VM interfaces. Kirk McKusick and Mike Karels of UC Berkeley, CSRG, spent several days discussing the issues with us. The other members of Sun's System Software group gave considerable assistance and advice during the design and implementation of the system.

10. References

- [ACCE 86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.
- [AT&T 86] AT&T, *System V Interface Definition*, Volume I, 1986
- [BOBR 72] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10", *Communications of the ACM*, Volume 15, No. 3, March 1972.
- [GING 87] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.
- [JOY 83] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.
- [KEPE 85] Kepecs, J. H., "Lightweight Processes for UNIX Implementation and Applications", *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.
- [KLEI 86] Kleiman, S. R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.
- [MKCU 84] McKusick, M. K., W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", *Transactions on Computer Systems*, Volume 2, No. 3, August 1984.
- [MURP 72] Murphy, D. L., "Storage organization and management in TENEX", *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.
- [ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

CAS Perspective on the Maturation of UNIX

Susan A. Funk
Chemical Abstracts Service
P. O. Box 3012
Columbus, Ohio 43210
(614) 421-3600 Ext. 3316
ucbvax!cbatt!osu-eddie!chemabs!saf27

ABSTRACT

Chemical Abstracts Service (CAS), which traditionally has relied on large mainframes for computing support, installed its first UNIX system in 1980. The first applications of UNIX were general business and administrative support and software development support. CAS was attracted to UNIX by its reputation for portability, flexibility, and ease of development. End user reaction has been generally positive and usage has increased in the past six years so that thirteen VAX 11/785's now support over 1100 users.

More recently, CAS has begun to extend its UNIX design and development efforts to include more traditional business applications. The integration of UNIX into a mainframe-oriented environment has been difficult and, at times, controversial. Staff who rely on the support facilities provided by other operating systems have been reluctant to endorse and accept UNIX. While retaining a commitment to UNIX and recognizing its unique qualities, CAS has identified certain areas in which it believes the UNIX operating system must mature. Some of the areas of concern are backup/recovery capabilities, performance analysis, job scheduling, operator support, security, and disk space management.

This paper presents an overview of the CAS experience and a discussion of the capabilities that we believe must be considered and addressed if UNIX is to provide effective support for large-scale business applications.

1. Introduction

Chemical Abstracts Service, a division of the American Chemical Society, serves the worldwide scientific community by providing information services that guide scientists and engineers to the chemical information they need. Those services include printed products such as the weekly publication *Chemical Abstracts*, which is a complete guide to the world's chemical literature, and online information services such as STN International. As the volume of technical and scientific information has grown exponentially, CAS has been a leader in the application of computer technology to scientific information processing and computer-based publishing. Over a period of years, the Information Systems organization has developed a large base of highly complex and customized software to support CAS products and services. This includes software to build and maintain several large publication databases, to provide a chemical substance identification system, and to format and photocompose CAS publications using an in-house APS-5 phototypesetter.

Most of this software has been developed for use on large IBM mainframe systems. Today CAS development efforts focus on techniques and software to provide faster and more effective processing, searching, and dissemination of chemical information. Currently our data processing center contains an IBM 3081K+ and an IBM 3090 Model 200, both running the MVS/XA operating system. Information Systems has a large staff, consisting of over 350 programmers, systems analysts, information scientists, engineers, technical support and operations staff.

2. The Arrival of UNIX

In 1977 CAS first experimented with online interactive support for software development by introducing several IBM TSO (Time Sharing Option) terminals. TSO allowed programmers to edit code online, to execute certain dataset utilities interactively, and to perform online testing and debugging of code. Most programmers responded to TSO enthusiastically, and the company gained noticeable improvements in productivity.

By 1980, problems had arisen. Contention for the limited number of TSO terminals was heavy, often causing long and frustrating waits for the staff, and response time was degrading. It had become apparent that the mainframe processor, an IBM 370/168 AP, could not continue to provide adequate support for the increasing development and production load being placed on it. Several alternative solutions to the problem were considered, resulting in a decision to distribute a part of the development workload to a minicomputer on an experimental basis. After careful evaluation, a PDP 11/70 running the UNIX operating system was installed. UNIX was chosen primarily because of its reputation for portability and the large set of tools it provides for end users. The particular implementation of UNIX that CAS selected was IS/1, a product of INTERACTIVE Systems Corporation. An evaluation period with a small group of programmers and software designers proved successful, so additional systems were ordered.

At the same time that CAS was seeking an alternative to TSO for online programming support, management had recognized the potential benefits of automating typical office support tasks and was considering hardware and software alternatives. Once again, UNIX was a leading contender for this function due to its portability and flexible set of tools. Also, there were obvious benefits to be gained from using the same operating system to support both office automation and software development functions. An additional machine was acquired and installed for administrative support.

Most of the software development staff members made the transition to UNIX quite easily. They learned shell programming quickly and soon libraries of personal and project tools proliferated. Non-technical staff were slower to make the adjustment, as is to be expected. Many of them found the commands difficult to learn, the documentation lacking in examples, and the system messages uninformative. However, as they became more familiar with UNIX many of the complaints subsided.

In addition to text and source code editing and document processing, the primary use of the systems was for electronic mail within CAS. A link was also established with the American Chemical Society (ACS) office in Washington, D.C. to allow electronic mail traffic between the Columbus and Washington

sites. This has been a significant time saver for those who need to communicate frequently with staff in Washington. A Remote Job Entry (RJE) link was established to the IBM mainframes to allow transfer of source code and submission of jobs in batch mode from UNIX terminals. The RJE link also allows users to transmit text files for printing on the XEROX 9700 laser printer which is connected to the mainframe.

3. The Current Environment

In the seven years since the first UNIX machine was introduced, the CAS computing environment has changed dramatically. UNIX-based systems now constitute a large part of the computing facility. There are currently thirteen VAX 11/785's installed. In 1983 the operating system migrated from IS/1 (V7-based) to IS/3 (System III based). In 1986 IS/3 was replaced by ULTRIX 1.2. The user population numbers over 1100, of which perhaps half could be categorized as frequent users. Each staff member involved in software development, including managers and support staff, now has a terminal at his or her own desk. Many other staff, both clerical and professional, have dedicated terminals and the demand for terminals continues to grow.

Several benefits have been derived from the addition of UNIX to the environment. It has allowed users to gain greater access to computing support and to gain more control over their own information needs. Many of our non-technical users have become knowledgeable enough to develop their own tools to automate routine tasks - tools that they might never have been able to develop otherwise. The development staff have found it substantially easier to develop online applications under UNIX than under MVS. Another benefit is the great flexibility available in configuring system components, including the support for a wide variety of terminals. The combination of UNIX and smaller processors has also allowed CAS to increase computing capacity in smaller, more cost-effective increments.

As the popularity of the UNIX systems has grown and users have become more knowledgeable, new applications have been proposed for development. During the past year, efforts have been underway to install several new business systems based on a commercially available accounts receivable package. Soon a complex of UNIX-based machines will be used to support searching within our online search system, STN International. Also, an online bulletin board system for some members of the American Chemical Society is currently being developed.

4. UNIX Issues

Much of the discussion concerning the viability of UNIX in the business world has centered on three issues: the need for a standard version of UNIX, the need for a more user-friendly interface, and the lack of application software on the market. Certainly each of these issues warrants attention. However, based on our experiences, CAS suggests that there is another area in which UNIX must mature if it is to gain acceptance as an operating system for business applications; it is the area of operational services. While CAS development staff have generally given UNIX favorable reviews as a base for application development, others in the company have been reluctant to endorse it. This reluctance is primarily because UNIX does not provide some of the basic services that users and operations staff have been accustomed to and relied on in the past. Several of these services are discussed here.

4.1 Backup/Recovery

Over the years CAS has evolved a satisfactory system of nightly incremental and weekly full backups based on UNIX utilities. Until now this has provided adequate protection against the loss of data. However, with the installation of new applications and the addition of large Winchester disks to our systems, our backup policies and procedures are being re-evaluated.

One concern that has escalated over time is that UNIX provides only primitive support for magnetic tape drives. For example, the UNIX device driver does not contain any logic to detect the end of the physical media. Instead, all logic to process an end-of-reel condition must be provided by application programs such as "tar" or "dump". This was brought to our attention rather forcefully recently when a large file was lost and an attempt was made to recover it from an incremental dump. However, the dump, which had been created by piping cpio output to the device driver, was incomplete. The data had exceeded the capacity of a single reel and the remainder was silently lost.

With nightly file backups being performed on thirteen machines, the issue of tape reel management has become a significant concern. There are no UNIX facilities for machine readable labelling of tapes; therefore, manual labelling and record-keeping is required for the 635 tapes in our backup pool. This manual effort is cumbersome and inevitably leads to errors, such as mislabelling and mounting of incorrect tape reels. It can also lead to the loss of critical data if a tape is accidentally overwritten. There is also a need for an automated index to tapes which would include such information as creation date and expiration date.

When running applications that perform extensive processing against large files, users have expressed a need for some form of restart capability. At the mainframe a facility is provided whereby the application can request a checkpoint to be taken at certain specified intervals. If there is a program failure or a system failure before completion, the application can be restarted at the most recent checkpoint rather than having to be completely rerun. This can provide a very significant savings in elapsed time, CPU time, and human effort.

4.2 Security

Security has been a topic of particular concern at CAS for some time, as it is within many other installations. CAS has placed a high priority on protecting the data within its computer systems from access or modification by unauthorized persons, and has taken steps to enforce protection at the mainframes. To this end, a vendor-provided package has been installed on the mainframes which interfaces with the operating system. Security is controlled by a small set of staff members who are authorized as Resource Security Administrators. Data is considered to be protected by default, that is, those users and processes which are allowed access to a file must be specifically identified to the system. An access rule must exist for each file created. All access rules are defined by the Security Administrators. Access can be granted on the basis of individual login names, by group, by functional unit, or by physical site.

The UNIX approach to security contrasts dramatically with that in the mainframe environment. File security is oriented toward a knowledgeable

user population in which information is frequently shared. Each user is given responsibility for the protection of his/her own files. This requires that each user has a thorough understanding of file protection modes and how to modify them. Users must also understand that even protected files are open to the Superuser unless they are encrypted. Our experience has been that it is very difficult to educate all users adequately on this topic. Users who routinely handle certain categories of confidential information are careful to set proper protection modes. However, other users may sometimes leave sensitive files unprotected due to misunderstanding or simple forgetfulness.

The fundamental question of a protected versus an open environment has been a topic for debate among CAS staff. While security consultants and many management personnel urge a change to a "protection by default" mode of operation, many users favor retaining the current default. The sharing of files among users is common, whether the files contain document text, source code, or executable binary data. Some users have suggested that a useful compromise would be to allow protection masks to be specified on a directory basis, for example, to request that each new file created in a specific directory should be assigned a mode allowing read-write access only by the owner.

The three level approach to file protection (owner, group, all) used in UNIX has proven to be overly restrictive when applied to a large, complex user population. The number of groups which potentially could be defined within that user population is very large and quickly becomes unwieldy from the administrative standpoint. Some versions of UNIX set unreasonably low limits on the number of groups to which a user may belong. This may lead to improper protection of files or directories as users choose to bypass the process of defining a new group for the purpose of sharing access to a file or group of files and simply make the file(s) accessible to everyone.

At the mainframes, detailed records are kept of activities related to file access. Any change to the file access rules by a Security Administrator is recorded on a report. For data which has been defined as sensitive or confidential, each access in read or write mode is also entered into a report. At any time, a trace can be enabled for a particular login so that all file accesses under that login are recorded. The reports are reviewed daily by the Security Administrators and any unauthorized attempts to access data are given immediate attention. The reports are maintained for a period of time so that problems can be traced. UNIX, on the other hand, provides little or no tracking of the activities of privileged users, particularly the Superuser. Although CAS Superusers are limited to a small, well-controlled group, this lack of audit trails has been a continuing cause for concern. It allows a user who knows the root password to review or delete any file on the system, to change file ownerships, or to change modes without leaving an audit trail which can be traced.

UNIX does not provide a facility for monitoring and recording login attempts, especially those that are unsuccessful. This was considered essential, so code was added to perform this function in spite of the fact that CAS generally refrains from modifying vendor code. Every login attempt is monitored and pertinent information about unsuccessful attempts is recorded in a file. After several unsuccessful login attempts the user is prohibited from logging in even if the correct password is entered. The list of unsuccessful attempts is automatically sent to the system administrators for followup.

Users accessing the mainframe are required to modify their passwords at regular intervals. In the past CAS made use of the UNIX password aging facility to enforce the same requirement. However, when the conversion to ULTRIX took place this capability was no longer provided. This was also considered an essential feature, so once again CAS staff added the necessary code to set and enforce password expiration dates.

4.3 Scheduling of Jobs

Because of the online timesharing orientation of UNIX, execution of jobs in a batch mode is awkward. Although most of our UNIX-based applications developed up to this point have been intended as online applications, there are still some functions which can be performed in a background (batch) mode. Recently we have experienced an unanticipated demand from end users for a batch job submission capability. Often the end users of applications do not need or want to be involved in the routine execution of the tasks. These components may require execution at some specified time interval, or merely at the user's request. They may require large amounts of elapsed time to run to completion. In the mainframe environment, where batch processing is a significant component of the total processing load, these applications are scheduled by users through the Computer Operations department. Each day the jobs are entered into a series of queues by the operators and started automatically at a designated time of day or upon completion of another related job. The operators at the main console can, at any time, check the status of the job queues and adjust priorities. They are notified as each job starts and terminates, whether successful or abnormal. Operators are then responsible for notifying maintenance programmers or the end users when a failure occurs. A hardcopy log of activity is routed to the end user for each program that is run.

UNIX does not provide any convenient mechanism to queue a set of jobs and to define dependencies among them. The "cron" function allows programs to be executed at certain times of the day, but it is better suited for execution of routine system administration tasks. It does not provide the flexibility and ease of use needed to process a queue of jobs which changes on a daily basis.

Because of user demand for a capability of this type, CAS is in the process of defining and implementing an experimental Job Submission Tool which will allow operators to submit, schedule, and monitor jobs run in an offline mode. It will collect statistics and status information as the job runs and produce a report at job termination. It will also write status messages and completion codes to the system console.

4.4 Operational Support

All CAS mainframes and minicomputers are located in one central computing facility which is staffed around the clock by trained computer operators. The operators control and constantly monitor system activity at the mainframes. They are immediately made aware of any problem situations which arise and are responsible for taking the necessary measures to resolve the problems. In contrast, the UNIX-based systems have been treated essentially as turnkey systems since their introduction, running largely unattended. It has been assumed that users are aware of and responsible for any problems with their own applications. Overall system operation has been the responsibility of a system administrator who does a periodic status check or reacts when a problem is reported by a user.

With the growth in the number of UNIX machines in our installation and the processing loads placed on them, it has become more important to monitor system status and to be able to react quickly when system failures occur. Operations staff have found, however, that UNIX does not provide a very useful set of tools for determining what is happening at any given time. A frequent complaint is that system diagnostics are brief and do not provide the operator with enough information to determine the course of action to take. In contrast, each mainframe diagnostic includes an error number which can be looked up in a reference manual to obtain specific information on the cause of failure and action to be taken. Often a systems programmer must be called upon to determine whether a UNIX error is due to hardware or software failure, increasing the amount of system downtime. (System V has made progress by providing an error message manual, but further improvement is needed.) Generally, the systems programmer must rely on past experience to interpret the message and diagnose the problem. Sometimes he/she must refer to the operating system source code in order to pinpoint the problem. Transient hardware failures are often particularly difficult to locate, due to non-specific error messages.

Operators and system programmers have expressed concern that there is no comprehensive mechanism for automatic logging of system messages. Messages appear only on the hardcopy at the console and are not maintained online. In contrast, MVS maintains a history of such information as hardware and software error statistics, job starts and stops, tape mounts, application-generated console messages and operator replies, and error messages. This information is a valuable aid to staff who need to trace through a particular problem, allowing them to view the information online and search it to locate specific events.

4.5 System Accounting

Soon after the first UNIX system was installed, it became apparent that the operating system did not produce a large amount of useful information regarding system/user activity and system resource utilization. As we have upgraded to newer versions more data has been provided, but improvement is still needed. It is essential for any business to make the most effective use of its computing power. CAS monitors utilization of all hardware carefully and has established service level objectives for response time and system up time. The ability to analyze trends and forecast the need for additional processing power is critical, especially for an application such as our online Search Engine Complex which supports a key component of our business. Accurate resource utilization data is also required in order to perform load balancing among the thirteen VAXes. Although UNIX provides much of the raw data needed to analyze system use and performance, our experience shows us that it provides only minimal support for manipulating and analyzing the data. Proper interpretation of the statistics often requires either the services of a UNIX "guru" or a detailed review of the source code to determine what the metrics actually represent. The production of meaningful reports, especially those suitable for management level review, usually requires some software development effort.

A capability that has recently received some attention is that of allowing the chargeback of CPU cycles to projects or organizational units. The UNIX accounting system is based on an assumption of accounting by user. Typically,

however, our users are involved in several activities at any one time, each of which may have a different funding source. Currently, there is no way that fees for resource use on the UNIX machines can be accurately distributed among projects.

4.6 Disk Space Management

In the mainframe environment, file space requirements must be predefined and limitations are enforced by the operating system on a per file basis. UNIX frees the user from those restrictions. This freedom is one of the advantages of UNIX, yet the simplicity of the approach means that space management is not as controllable or predictable under UNIX. Consequently, we find ourselves functioning primarily in a "reactive" mode. During the course of a day it is difficult to monitor file space utilization adequately on all systems. A problem is detected only when a file system approaches 100% utilization or has actually reached it. Even routine monitoring, on an hourly basis for example, cannot always guarantee enough advance warning to avert a problem situation. A single looping process can very quickly use all available space on a file system, totally degrading system response and leaving users of the impacted file system unable to function. Uncontrolled consumption of file space by a single user or process impacts all users of that file system and can sometimes lead to critical data loss, particularly for applications running in background mode. Under MVS, only the particular user or job is affected by excessive (unplanned) file space consumption. UNIX has no file overflow mechanism in contrast with MVS, which allows a secondary allocation to be specified. When that secondary allocation is used, the staff who monitor DASD management are notified. They can then take steps to increase the available space or move the file before a critical shortage occurs.

4.2BSD and its derivatives do offer the disk quota capability for regulating usage by individual users, but defining and establishing realistic quotas for a large user population requires substantial administrative effort. In addition, the use of quotas can limit flexibility in an environment where a group of users need to create/update a common set of files. For these reasons, CAS has chosen not to implement a quota system.

4.7 System Stability

The stability and reliability of the systems are a critical yardstick when measuring suitability for day-to-day production use. It is imperative that the maximum system up-time be achieved. CAS has defined specific system availability objectives for each of its processors. In general, we have found that the UNIX systems tend to be more sensitive to failures. A failure often results in an entire system being out of service for some period of time. Even if that period is brief, it may have serious impacts on user productivity and schedules. In addition, any system failure raises concerns about data integrity. In contrast, the mainframe operating system has more built-in error detection and recovery mechanisms that allow continued operation in a degraded mode, under certain conditions. As an example, the occurrence of a bad sector on one of our Winchester disk drives is likely to result in a system crash and several hours of system downtime; however, some other operating systems would handle the error recovery transparently.

5. Conclusion

Although many comparisons have been made between MVS and UNIX in this paper, it is not meant to imply that we believe UNIX should evolve into MVS. We understand the origins of UNIX and the underlying goal of simplicity that has governed its development. We recognize the batch job origins of MVS and the contrasting interactive orientation of UNIX. We also recognize that, unlike the proprietary operating systems with which many businesses are most familiar, UNIX has strived to remain hardware independent in order to achieve maximum portability.

The purpose of presenting these comments to this forum is twofold. One is to share experiences and insights with others who may be considering or may already have begun to integrate UNIX into a large business installation. I hope that they might consider some of the issues raised at CAS and plan carefully to address them in advance. The other purpose of this paper is to raise the consciousness of those who are actively involved in development of the UNIX operating system and UNIX-based applications. Although some of these issues may seem irrelevant in a small business installation or in an academic environment, they are important considerations in a large business where system reliability and user satisfaction are measures of success. Users who have been accustomed to a particular level of service, including certain safeguards and facilities, will continue to expect the same service regardless of the underlying operating system.

Some of the difficulties that CAS has encountered seem to result from the UNIX orientation toward a small installation where a single system administrator is responsible for overall system operation. However, in a large installation like ours, the environment is very structured. Functions typically handled by a system administrator are distributed among several people in different operating units, each with its own specific set of responsibilities.

The disparity between versions of UNIX also contributes to the difficulties. Having used both AT&T and Berkeley based implementations, the differences have been very noticeable. For example, ULTRIX (Berkeley) provides the disk quota capability for file system management, while AT&T does not. AT&T provides password aging for security, while Berkeley does not. Some of the differences are very obvious, others are more subtle. Neither version seems to have a clear advantage for business applications.

Some of these problems could be alleviated or resolved through development of software by the using installation. However, such development brings an additional expense that must be considered when making the decision to move to UNIX. The expense is difficult to justify when the required capabilities are readily available under other operating systems. It may be difficult to convince management that UNIX has enough advantages to outweigh the level of effort required to implement and support it.

There are several other alternatives to resolving these issues. One is for developers of the UNIX operating system to add enhancements, either via modifications to the operating system itself or by adding new utilities. Another option is for application package developers to provide new software products. Recently some vendor products that provide enhanced support for such functions as backup and recovery have been introduced to the market. This is encouraging. However, we have reviewed several of these products and found

that they still tend to be oriented toward small installations. They are generally no more sophisticated than the procedures that have evolved through our experiences over a period of several years. An enhanced dialogue and increased sharing of experiences among users may also help to improve the environment. Meetings such as the recent USENIX Large System Administrator's Conference provide a useful forum for discussion.

Through our experiences, we have also come to the realization that we must begin to define some boundaries for UNIX application development. That is, rather than directing development of all new applications toward UNIX, we need to review each application individually and assess its suitability for operation under UNIX, particularly in light of some of the issues raised here.

BIBLIOGRAPHY

Robert Elz, "Disc Quotas in a UNIX Environment", *ULTRIX-32 Supplementary Documents Volume III*, Digital Equipment Corporation (1984).

Donna Hawrot and Les Comeau, "Facing Commercial Development", *UNIX Review*, Vol. 4, No. 11, pp. 46-56 (November, 1986)

IS/3 User's Manual, INTERACTIVE Systems Corporation (1983).

Carol Realini and Donal O'Shea, "UNIX in the Business Market", *UNIX Review*, Vol. 4, No. 11, pp. 24-29 (November, 1986)

Vanessa Schnatmeier, "Due Credit: UNIX Enters the World of Banking", *UNIX/World*, Vol. 4, No. 2, pp. 26-31 (February, 1987)

Steven D. Stamps, "Right for the Job?", *UNIX Review*, Vol. 4, No. 11, pp. 33-45 (November, 1986)

UNIX System User's Manual (System V), Western Electric Company (1983)

UNIX System Administrator's Manual (System V), Western Electric Company (1983)

ULTRIX-32 Programmer's Manual, Digital Equipment Corporation (1986).

ULTRIX-32 System Manager's Guide, Digital Equipment Corporation (1985).

MVS is a trademark of International Business Machine Corp.
ULTRIX is trademark of Digital Equipment Corp.

Keeping watch over the flocks by night (and day)

Kenneth Ingham
University of New Mexico Computing Center
Distributed Systems Group
2701 Campus NE
Albuquerque, NM 87131
(505) 277-8044
ingham%charon.unm.edu or ucbvax!unmvax!charon!ingham

Abstract

Over the last several years, the number of machines maintained by the University of New Mexico Computing Center has increased rapidly, yet the number of system managers monitoring these systems has remained static. Consequently, the system managers were faced with the task of watching more and more machines; since only one system manager is on call at any time (known affectionately as "DOC"), this soon proved to be an unacceptable situation. Shell scripts running every six hours gave some assistance; this was offset by the fact that the scripts generated a great deal of output indicating normal system operation, which the system manager still had to scan carefully for signs of trouble. This paper describes *watcher*, a flexible system monitor which watches the system more closely than the human system manager while generating less output for him to examine.

Running more often than the above mentioned set of shell scripts, *watcher* is able to keep closer tabs on the system; since it delivers only a list of potential problems, however, this extra monitoring produces *no* corresponding increase in the demand on DOC. No problems slip by unnoticed in the more concise output, leading to an improvement in overall system availability as well as the more effective utilization of the system manager's time.

0. Acknowledgments (I couldn't have done it without you)

I would like to thank Leslie Gorsline for her assistance in the writing of this paper. Without her, this paper might not have been. Also thanks to the UNMCC distributed systems group for their comments that helped improve *watcher*.

1. Background (the problem)

The computing facilities offered by the University of New Mexico Computing Center (UNMCC) include three microvaxen, five large vaxen (780 or bigger), and a Sequent B8000. In addition to these Unix/VMS machines, the UNMCC Distributed Systems Group (DSG) monitors a number of the various microvaxen and sun workstations scattered across campus. This duty falls to the DSG Programmer designated as "DOC", or "DSG On Call", who receives his beeper based on a monthly rotation schedule.

In the past, shell scripts running every six hours reported various system statistics to DOC, who then scanned the output for signs of possible trouble. The output of these shell scripts became overwhelming as the number of machines and potential problems grew; corresponding to this increase in output was an increase in the amount of time that DOC had to spend reading this output. In addition, most of this output merely indicated normal system operation; potential problems were buried amongst non-problems. Because of this, DOC could often waste a tremendous amount of time wading through system status reports, time which can be better spent actually fixing system problems.

Unix is equipped with many powerful tools for program development, but none which simply watch the system for signs of trouble. Programs like *ps* and *df* provide information regarding the current state of the machine, yet it still remains DOC's responsibility to interpret this information and assess the health of the system at any given time. This deficiency can be rectified by providing the system with the capacity to determine its own state of health, advising DOC when it notices a problem which requires DOC's intervention.

2. Design Goals (devising the solution)

In designing *watcher*, the author closely examined just what DOC does in monitoring the system; just how *does* DOC spot potential trouble in the DOC reports? These reports consist of output from *df -i*, *ruptime*, *ps -aux | sort*, and the tail of *cronlog*, which usually only changes in the middle of the night. It was determined that DOC's task consisted primarily of scanning various numbers in this output, deciding whether or not they had exceeded an allowable maximum or minimum, or if the values had changed too much from the last time the command was run, assuming the last value is even remembered. Getting a computer to do this is more complicated than might seem at first glance, due to inconsistencies in the location of pertinent information between runs of these commands. For instance, the process occupying the fifth line of *ps -ax* might next time appear on the eighth line; similarly, *uptime* does not consistently put germane information in the same place on the line.

While flexibility is certainly a primary design consideration, it is not the whole story. In order to improve DOC's effectiveness, the program should run frequently, roughly every two or three hours, catching problems early (hopefully before they have affected the

users). Thus, the program should also be as silent as possible except when it detects a potential problem; any advantage DOC gains in using *watcher* would be eliminated if the program delivered an exceedingly verbose status report every two hours. *watcher*'s problem reports should be exact and concise, leading DOC immediately to the trouble.

The problem of reducing the amount of output DOC must process can be approached in different ways, including the redesign of the current shell scripts. A simple *awk* script can watch the output from *df* [1]. However, each command would require a custom tailored *awk* script to look at it. This task grows more complicated as the number of programs running increases. While a program could be written to generate these *awk* scripts, this process is needlessly complex; for only a bit more work, an efficient C program such as *watcher* can be developed.

3. Design (actual implementation of the solution)

Run at intervals specified in *crontab*, *watcher* parses a control file (*./watcherfile* by default) with a *yacc* generated parser, building a data structure containing all of the information from the file. The file contains the list of commands *watcher* should run (the pipeline), output specifications for each command (the output format), and the guidelines used in determining if something is amiss and should be reported to DOC (the change format). A sample *watcher* control file would look something like this (comment lines begin with a '#'):

```
# Here is the pipeline and its alias:
(df -i | /usr/ucb/tail +2) { df }
# the output format; this is a column output format:
    $1-9 device%k $41-42 spaceused%d $64-65 inodesused%d:
# and the change format:
    spaceused 15%;
    spaceused 0 89;
    inodesused 15%;
    inodesused 0 49.

# another command example:
(/usr/ucb/ruptime | fgrep -f UnmIHosts) { ruptime }
# this is a relative output format
    2 status%s 1 machine%k 7 loadav%d:
# and another change format:
    loadav 0 10;
    status "up".
```

The first entry causes *watcher* to run the *df* pipeline listed in parentheses. When reporting problems, *watcher* refers to this command by the alias provided in the braces; if no alias appears, *watcher* uses the entire pipeline.

The output format instructs *watcher* how to parse the output; column format, indicated in the output format by **num-num**, instructs *watcher* that the output should be parsed by columns, while relative format, denoted by a single integer, shows that the output should be broken up by whitespaces. Through the convention **name%type**, the output format also names each field, indicating whether the field is numeric, string, or keyword, specified by **d**, **s**, or **k** respectively. Keyword fields are used to match up corresponding output lines between runs. Thus

```
41-42 spaceused%d
```

indicates that this field, named **spaceused**, contains numeric information in columns 41-42, while

```
2 status%s
```

informs *watcher* that the second word (group of non-whitespace characters) on the line is a string field named **status**. For the *df* example given above,

```
Filesystem  kbytes  used  avail capacity  iused  ifree %iused Mounted on
/dev/hp1f   52431  39763  7424   84%      6937   9447  42%    /develop
```

device would be */dev/hp1f*, **spaceused** would be 84, and **inodesused** would be 42. Similarly, the output from the *uptime* example, which looks like this

```
charon      up 26+07:53,  17 users, load 3.12, 2.90, 2.66
```

would be broken at the following places:

```
charon | up | 26+07:53, | 17 | users, | load | 3.12, | 2.90, | 2.66,
```

assigning "up" to **status**, and 3.12 to **loadav**.

The name field also appears in the change format, designating allowable values for this field to have. These values can be specified as single character strings in the case of string fields; in the case of numeric fields, the values take the form of either percentage or absolute changes, or a minimum and maximum which delineate an acceptable range. Thus

```
inodesused 15%;
inodesused 0 49.
```

signifies that DOC should be notified if the field named **inodesused** increases by more than 15% from the last run, or if it is outside the range 0 to 49; similarly

status "up";

informs *watcher* to notify DOC if the **status** field contains anything other than the word "up".

As *watcher* parses the output of a pipeline, it stores the pertinent parts of the output in a history file (by default, *./watcher.history*). The next time *watcher* runs, it reads this file to provide comparison values for the command. If a command is new (i.e. it has no previously-stored output in the history file), *watcher* checks the fields which require no previous data, such as min-max fields, while still storing *all* of the relevant information to the history file. Thus, the next time the new command is run, it will be an *old* command, and meaningful between-run comparisons can be made.

When *watcher* detects no problems with the system, DOC receives an empty mail message with the subject "*hostname* had no problems at *date*"; this is to insure that *mail* is running correctly. When it notices a problem which should be brought to DOC's attention, it mails the system problem report in a concise format, explaining what is wrong and why. Thus, rather than the megabytes of shell script output that DOC used to receive and have to read, he merely sees this when he reads his mail:

```
Mail version 5.2 6/21/85. Type ? for help.
"/usr/spool/mail/ingham": 5 messages 5 new
N 1 root@charon.unm Sat Apr 11 16:00 8/212 "charon had no problems at Sat"
N 2 root@ariel.unm Sat Apr 11 16:00 8/208 "ariel had no problems at Sat "
N 3 root@geinah.unm Sat Apr 11 16:00 11/417 "System problem report for gei"
N 4 root@izar.unm Sat Apr 11 16:00 8/204 "izar had no problems at Sat A"
N 5 root@deimos.unm Sat Apr 11 16:00 8/212 "deimos had no problems at Sat"
```

The letters indicating no problems can be immediately deleted, and DOC can turn his attention to the letter indicating a system problems. A sample problem report would look something like this:

```
df has a max/min value out of range:
/dev/hp0h 140488 111195 15244 91% 10145 28767 26% /usr
where spaceused = 91.00; valid range 0.00 to 89.00.
Also it had inodesused change by more than 10%.
Previous value 20.00; current value 26.00.
```

Note that if a line has more than one indication of a problem, all anomalies are included in the report. This provides DOC with as much information as possible, allowing him to determine the problem quickly and devise a rapid fix (hopefully before users know something is amiss).

4. Results (how its helped us)

watcher's primary advantage lies in the reduction of DOC's work load. It has taken over the more menial aspects of monitoring a system, tasks like reading and comparing numbers, giving DOC more time to concentrate on bugs of a nature which *watcher* isn't set up to monitor, such as problems in the accounting system. DOC is apprised of potential problems quickly, and in some cases can repair them in less time than simply reading the shell script output would have taken.

The ability to monitor changes between runs has also helped bring to our attention some problems which were missed in the DOC reports. For example, disk space on */u2* on one of our machines jumped by more than 15%. Since this jump did not force the total space used above 90%, at which point DOC would have investigated the filesystem, it is unlikely that DOC would have even noticed this sudden change. The facility to watch for relative changes between runs enables DOC to catch problems in their infancy, and fix problems such as filesystems filling up too rapidly before they inconvenience the users.

Since the system manager specifies not only the commands *watcher* will execute and the time lapse between successive runs, but also the parameters which indicate system anomalies, *watcher* can easily be seen as a very flexible, general system monitor. Its use at UNM has provided an increase in the productivity of the system manager, which has led in turn to the increase in the reliability and availability of the systems at UNMCC.

5. Availability (how to get one)

watcher will be sent to the moderator of mod.sources after the conference is over.

6. References (you might also find this interesting)

- [1] Monitoring Free Disk Space, Rik Farrow, Wizard's Grabbag, *Unix World*, Vol. IV, no. 3, pp. 86-87.

X.400 Messaging on UNIX†

Andrew Draskoy
Gerald Neufeld

EAN Research and Development Group
University of British Columbia

ABSTRACT

The CCITT X.400 Message Handling System is becoming an accepted standard for electronic mail internationally. This paper examines the issues involved in implementing and using X.400 on UNIX. The EAN software developed at the University of British Columbia will be used throughout as an example of how these issues were handled in an actual implementation.

1. Introduction

The X.400 series of recommendations[1] describe a standard for Message Handling Systems (MHS). The MHS model includes a collection of User Agents (UAs) and a Message Transfer Service (MTS). An MTS comprises a number of Message Transfer Agents (MTAs). Operating together, the Message Transfer Agents relay messages and deliver them to the intended recipient User Agents using the underlying Reliable Transfer Service (RTS). The User Agents allow the users to create, edit, send, and receive messages. The Reliable Transfer Service consists of the session, transport, and network layers of the Open Systems Interconnection Reference Model[2]. Figure 1 shows the MHS layer model and indicates the protocols used to communicate between instances of a layer.

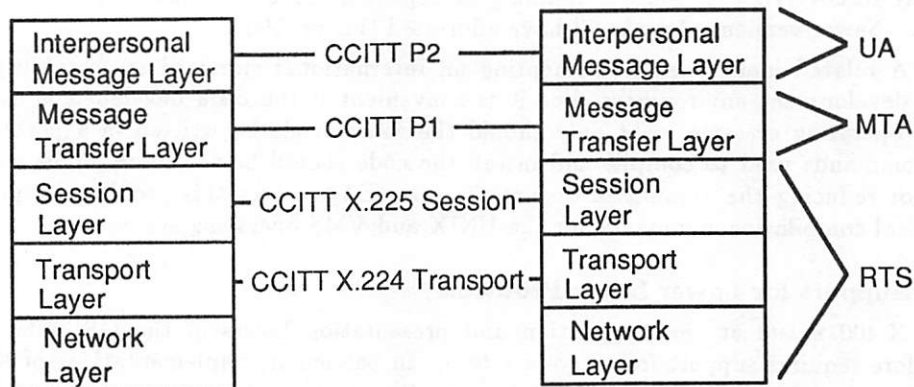


Figure 1 - X.400 Layers

† UNIX is a trademark of AT&T Bell Laboratories

2. Why X.400?

UNIX systems already include messaging software, namely */bin/mail* and *uucp*. There are also several packages available to add routing and RFC 822[3]-style addressing, including *sendmail*[4] and *smail*[5]. Despite the fact that UNIX users have enjoyed the use of a distributed mail system long before international standards existed, there are still reasons to implement X.400 on UNIX. A primary reason is that many computer manufacturers, national PTTs, and public messaging services have announced plans to provide an X.400 service. It seems likely that in the near future X.400 will be in widespread use, especially in Europe, where the CCITT recommendations are taken very seriously. These systems make up an increasingly large part of the "global mail network". If the UNIX community is to continue to be a large part of that network, it is necessary to have X.400 implementations or gateways on UNIX.

Another reason to implement any protocol standard is that it offers certain technical advantages. In this case, X.400 has desirable functions that mailers available on UNIX do not have. These include provision for multi-part and multi-media mail; delivery and non-delivery reports; probes; application independent message transfer system; receipt and non-receipt notification for inter-personal mail; and deferred delivery.

3. Implementation

The EAN message system[6] is an X.400 implementation developed starting in 1983 by the Distributed Systems Research Group within the University of British Columbia's Computer Science Department. It was written in C on 4.2BSD UNIX, and has been ported to several other operating systems, including VMS*. It is now used within the Canadian Research Network (CDNnet) and at many universities in Europe.

X.400 has several attributes that can potentially make its implementation on UNIX difficult.

3.1. Development Environment

An X.400 implementation, such as EAN, comprises a large amount of source code. This means that there must be an efficient means of handling source code that is distributed over many directories in several versions. UNIX provides a reasonable development environment, thanks to tools like *make* and *RCS*. Nevertheless, when working with such a large amount of code, problems arise. During the development of EAN, which currently has over 50,000 lines of code, it was quickly discovered that *make's* handling of dependencies across directory boundaries was inadequate. Newer versions of *make* [7] have addressed this problem.

A related issue when implementing an international standard is portability. UNIX has a good development environment, but it is convenient if the code developed is easily portable to other operating systems. Not only should the source code be written in a portable fashion, but the commands used to compile and install the code should be the same. This is for convenience and for reducing the amount of documentation. EAN solves this problem by providing its own identical compilation commands for the UNIX and VMS operating systems.

3.2. Support for Lower Layer Protocols

X.400 exists at the application and presentation layers of the OSI reference model and therefore requires support from layers 1 to 5. In particular, implementations of the session, transport, network, and link protocols must exist. Few such implementations are currently available.

X.400 specifies use of X.225 and X.224 for the session and transport protocols respectively. At least one reliable network layer protocol (and the layers below it) must be available. The X.400 series does not require a specific protocol to be used for the network layer, although it does state that the layer must be reliable and be able to interface to X.214 transport class zero (TP0). The only CCITT recommendation fitting this description is X.25[8]. With the addition of an

* VMS is a trademark of Digital Equipment Corporation

extra interface layer, it is possible to use other network protocols with TP0. EAN uses TCP/IP, X.25, DECNET, TTXP (a proprietary asynchronous protocol based on MMDF[9]), as well as other network layer implementations. From these examples it can be noted that protocols normally used to provide services covering not only the network layer, but layers above it, can be treated as network layer implementations. TCP/IP, for example, is a transport protocol, but is used as a network protocol by EAN.

Unfortunately, most variants of UNIX do not come with a reliable network level protocol implementation¹. The protocols used within *uucp* might do the job, but those routines are not directly available to other programs (assuming that it is undesirable to require users of your software to have UNIX source licences.) In addition, the *uucp* protocol is undocumented.

Implementing network protocols can be a major task. For some, such as the X.25 implementation added to Berkeley UNIX for EAN, it is necessary to place the implementation into the kernel. The UNIX kernel is not a preferred environment for program development.

3.3. Use of Binary Protocols

Mail systems in common use on UNIX tend to use text-based protocols. Such protocols will often encode envelope information as English character strings. Data which is numeric in nature must be converted to strings on generation and back to numbers on reception. Representation of complex data structures, such as those involving sets and sequences of various data types, requires a complex syntax.

The use of text-based protocols makes sense on UNIX due to the very powerful text-processing tools available. Files used by the message system may be easily accessed, making it easier for programmers and users to understand what is happening within the mail system. Parts of the mail software itself can be easily implemented using shell scripts and various tools and utilities.

In contrast, X.400 uses binary protocols which make it difficult for humans to read. This also prevents the implementor from taking advantage of one of UNIX's main software development strengths, shell programming. These problems can be partially overcome with the use of complex parsers, although this would be extremely inefficient.

The most obvious storage format for X.400 messages is X.409 encoding. This limits the users' access via existing UNIX tools, which are mostly text-oriented. In particular it is hard to treat X.409 encoded messages as text files that can be easily manipulated. A binary format seems inescapable given the fact that X.400 messages may contain not only textual information, but also multi-part and multi-media content such as FAX, teletext, voice, and forwarded messages. In addition, not all the text within messages is necessarily ASCII. T.61 is occasionally used as well. Although such a format causes setbacks, it also allows for efficient manipulation of the data, since there is no need to do the text parsing and generation that might otherwise be necessary.

3.4. Reliability

There are further problems with storing data on UNIX that affect messaging. Manipulating large message databases efficiently requires much correlated data. To ensure the integrity of the database, it is often necessary to be sure that data has been written to disk. In addition, the RTS supports a checkpointing facility that requires that data received prior to the issuing of a checkpoint acknowledgment be secured on disk. This allows a restart on the next failure, such as a reset at the X.25 level. The only way to ensure this on UNIX is using the *fsync* system call. Unfortunately, the implementation of *fsync* on most UNIX variants is quite inefficient, especially with respect to large files.

¹ The TCP protocols provided in the Berkeley distribution are an exception, but outside the ARPA community these are generally used only on LANs.

4. Interacting with Other Messaging Systems

There are three typical scenarios. In each scenario the X.400 system must be able to communicate with existing UNIX messaging software.

- 1) The X.400 implementation is intended for use as a gateway² between existing messaging systems and the X.400 community.
- 2) A group of sites decide to switch entirely to X.400. In most cases, a transition period will be necessary for the changeover from the previous mail system. An interface between the two messaging systems is needed during that period.
- 3) Most UNIX sites have been using *uucp* or RFC 822 for some time, and have established a number of useful connections which would not lightly be given up even if the decision was made to use X.400 as the message system of choice.

4.1. Gateways

Conversion between messaging systems based upon differing philosophies is a non-trivial problem. Conversion between the X.400 protocols and those used by the ARPA Internet (which includes a large number of *uucp* sites), is the topic of RFC 987[10]. It has not currently been widely implemented, although it is expected that it will be. A similar gateway was implemented in EAN before the creation of RFC 987. This has been working successfully for several years.

4.2. Naming and Addressing

X.400 and RFC 822 have similar approaches to many of the problems of messaging. Naming and addressing are areas where the divergence is great enough to cause significant problems in gatewaying.

X.400 users are identified by Originator/Recipient Names (O/R Names). There are two forms. The most common identifies a user by specifying a set of *standard attributes* and optionally some *domain defined attributes* which uniquely identify the User Agent. Standard attributes include country, private domain name, organisation, a sequence of organisational units, and a personal name.

RFC 822 names are strings consisting of two parts and a delimiter. One is the *domain*, which is generally a dot-separated list of *subdomains* which represent a series of organisational units, an organisation, and a top-level domain roughly equivalent to an X.400 private domain name. The other is the *local-part* for identifying a user within a domain.

Naming and addressing are sufficiently complex, and the ideal far enough from the practical, that pragmatism has tended to dominate the actual interpretations of the schemes used. In the ARPA Internet, login names were used for *local-parts*, machine names for subdomains, and network names for top-level domains. Recently the domain naming scheme has changed to a network-independent, organisation oriented one.[11]

The development of a naming scheme is further complicated by the need to allow for backward compatibility with old schemes. This is the reason for *domain defined attributes* in X.400. RFC 822 does not specifically allow for the existence of other naming schemes. In practice, UNIX implementations of RFC 822 have to allow for other schemes. This is why addresses that supposedly conform to RFC 822 sometimes have a *local-part* containing another full address, often a *uucp* address which must be parsed in a different manner.

The EAN RFC 822 gateway approaches name conversion differently from RFC 987. This is largely because of the addressing scheme used by EAN. During the initial development of the system, the majority of other sites that messages could be exchanged with were using *uucp*, RFC 822 or derived protocols. X.400 allows for situations like this through *domain defined attributes*.

² The term *gateway* is used here to mean an entity performing the protocol conversion between two differing messaging systems. This is the standard use of the term in messaging, but it has different meanings in other aspects of networking.

EAN uses these to encode the equivalent of the RFC 822 *local-part* and subdomains. Only the top level domain is actually encoded using *standard attributes*. This approach considerably simplifies address conversion.

5. Conclusion

X.400 is beginning to play a significant role in messaging. If UNIX is to continue to be at the forefront of messaging research and development, use of X.400 within the UNIX community may be vital. Implementing it on UNIX, in a manner that supplies all the functionality of current mailers, and allows continued communication with them, is a large task requiring the development of much supporting code.

6. References

- [1] CCITT Study Group VII, Message Handling Systems, Recommendations X.400-X.430, October 1984.
- [2] CCITT Study Group VII, Open Systems Interconnection System Description Techniques, Recommendation X.200-X.250, Geneva, March 1984.
- [3] Crocker, D. H., Standard of the Format of ARPA Internet Text Messages, RFC 822, August 1982.
- [4] Allman, E., Sendmail - An Internetwork Mail Router, *UNIX Programmer's Manual*, 4.2 Berkeley Software Distribution, Vol. 2c, August 1983.
- [5] Seiwald, C., Smail 1.3, *Unix Programmer's Manual* Supplement, UUCP Project, November 1985.
- [6] Neufeld, G., Demco, J., Hilpert, B., and Sample, R., EAN: An X.400 Message System, *Proceedings Second International Symposium on Computer Message Systems*, Washington, D.C., September 1985.
- [7] Fowler, Glen S., The Fourth Generation Make, *USENIX Association Summer Conference Proceedings*, Portland, Oregon, 1985.
- [8] CCITT Study Group VIII, Data Communications Interfaces, Recommendations X.20-X.32, October 1984.
- [9] Crocker, D., Szurkowski, E., and Farber, D., An Internetwork Memo Distribution Capability - MMDF, *Proceedings of the 6th Data Communications Symposium*, November, 1979.
- [10] Kille, S. E., Mapping Between X.400 and RFC 822, RFC 987, University College, London, June 1986.
- [11] Postel, J., Reynolds, J., Domain Requirements, RFC 920, October 1984.

THE X TOOLKIT

THE STANDARD TOOLKIT FOR X VERSION 11

Ram Rao
ULTRIX Engineering Group
Digital Equipment Corporation
Merrimack, New Hampshire 03054
ram@decvax.dec.com or decvax!ram

Smokey Wallace
ULTRIX Engineering Group
Digital Equipment Corporation
100 Hamilton Avenue
Palo Alto, California 94301
smokey@decwrl.dec.com or decvax!decwrl!smokey

ABSTRACT

The primary design goal of the X Toolkit is to provide the base functionality necessary to build a wide variety of application environments. It is important that the design be fully extensible as well as support the independent development of new or extended components. This is accomplished by defining a few, easily used interfaces that mask implementation details from both applications and component implementors. By following a small set of conventions, it should be possible to extend the X Toolkit in new and, as yet, unimagined ways and have the extensions integrate well with existing facilities.

The X Toolkit is a library package layered on top of the X Window System* Version 11. This package extends the basic abstractions provided by X to support human interface construction. It does this primarily by supplying mechanisms for both inter-component and intra-component interactions as well as a reasonably complete and coherent set of sample widgets. (A *widget* is the combination of an X window and its associated human interface semantics.)

To the extent possible, the X Toolkit is "policy free". The application environment defines, implements, and enforces policy, consistency and style. Each individual widget implementation implements its own policy. However, if the X Toolkit is to be successful, it must allow but not encourage the free mixing of radically differing widget implementations.

1. Introduction

The X Toolkit design is the result of joint efforts by Digital Equipment Corporation, Hewlett-Packard Company and MIT Project Athena. The design is currently being reviewed by a number of corporations interested in the X Window System. The X Toolkit implementation is to be distributed with source code and documentation as part of the MIT X Version 11 distribution.

The X Toolkit library provides tools that simplify the design of application user interfaces. It assists application programmers by providing a commonly used set of user-interface widgets. It

* The X Window System is a trademark of MIT.

also lets widget programmers modify existing widgets or add new widgets. Therefore, by using the X Toolkit library in their applications, programmers present a similar user interface across applications to all worksystem users.

The X Toolkit consists of:

- A set of intrinsic mechanisms for building widgets
- An architectural model for constructing and composing widgets
- A set of sample widgets built with the above

The intrinsic mechanisms are intended for the widget programmer. The architectural model lets the widget programmer design new widgets by using the intrinsics or by combining other widgets. The application interface layers built on top of the X Toolkit include a coordinated set of widgets and composition policies. Some of these widgets and policies are application domain specific, while others are common across a number of application domains.

The X Toolkit library provides an architectural model that is flexible enough to accommodate a number of different kinds of application interface layers. In addition, the supplied set of toolkit functions are:

- Functionally complete and policy free
- Stylistically and functionally consistent with the X Window System primitives
- Portable across a wide range of languages, computer architectures, and operating systems

1.1. Underlying Model

The underlying architectural model is based on the following premises:

Widgets are X windows

Every user-interface widget is contained in a unique X window, with the X window handle serving as the widget handle. This allows standard Xlib window manipulation procedures to operate on widgets. Because windows in X are inexpensive, the impact on performance is minimal.

Information Hiding

The data for every widget is private to the widget: it is neither accessible nor visible outside of the module implementing the widget. All program interaction with the widget is performed by a set of messages defined for the widget. The avoidance of public data structures leads to better maintainability because changes to data structures have very local effects.

Widget Semantics vs. Widget Layout Geometry

There is a clear separation of widget semantics from widget layout geometry. Widgets are concerned with implementing specific user interface semantics. They have little say over issues such as their size or placement relative to other widgets. Mechanisms are provided for associating geometric managers with widgets. Such mechanisms facilitate composition of widgets out of other widgets in a recursive manner.

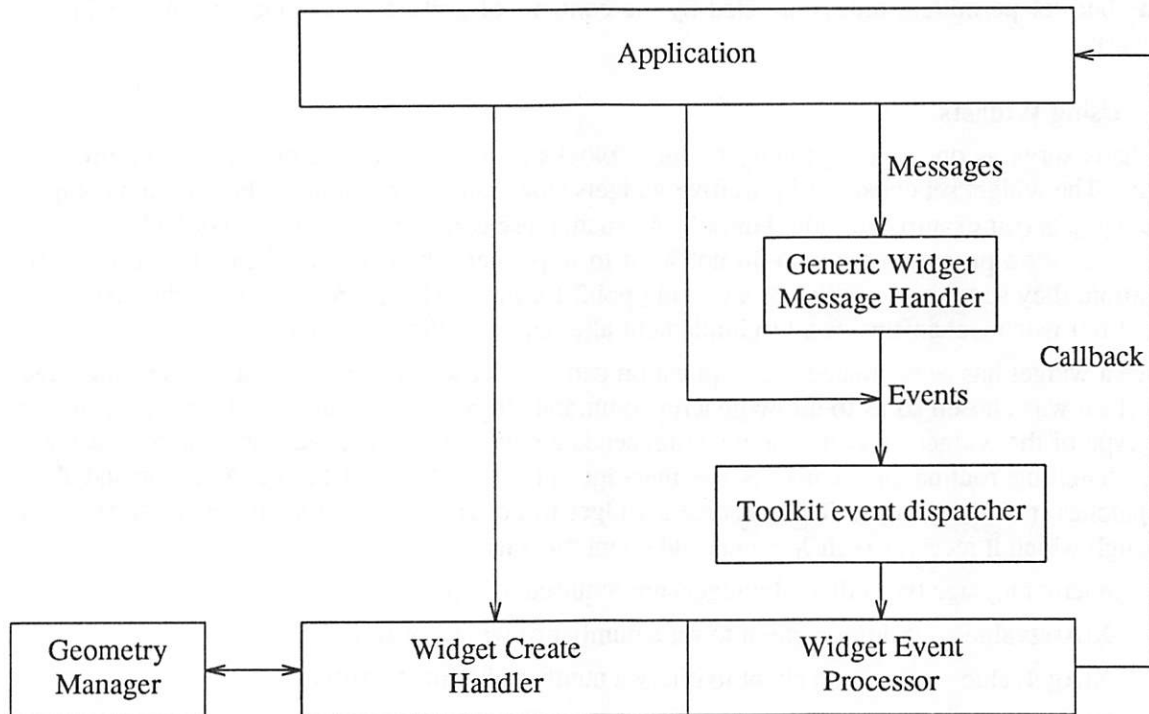
2. Widget Architecture

This section discusses the widget architecture from the perspective of a widget client. Widgets serve as one of the primary building blocks of the X Toolkit. Few applications will use the X Toolkit without using the supplied set of widgets. The widget set consists of primitive and composite widgets. For either primitive or composite widgets, however, the underlying architectural model is the same.

From the application's perspective all widgets provide a creation entry point, an event handler for X events and a message handler for application requests. Using this interface, an application program can create a particular widget instance and manipulate this instance through messages. However, from the widget's perspective the only application interfaces it provides are the creation entry point and an event handler.

With regards to all widgets, each widget instance is contained within its own subwindow. In many cases, this simplifies the code needed to implement a widget. In addition, because X provides a very rich set of window manipulation facilities, this also provides the application with a broad spectrum of widget manipulation abilities.

The following figure illustrates the interaction between an application and a widget.



2.1. Widget Creation

Each widget provides the application programmer with an entry point for creating a widget instance. This entry point has a common syntax across the full range of widgets within the X Toolkit. This syntax makes it easier for other pieces (for example, a composite panel widget) to create widget instances in a generic fashion.

A widget creation routine accepts three parameters: a parent window, a list of arguments, and an argument count. The argList, a variable length list composed of name/value pairs, contains information pertaining to the specific widget instance being created. These parameters provide a common widget creation syntax across the full range of widgets. When an application attempts to create a widget instance, it needs only to specify those parameters considered essential. The widgets provide reasonable defaults for all parameters. When a widget instance is successfully created, the widget identifier (window ID) is returned to the application. This identifier can then be used in message interaction with the widget.

An example of the creation entry point in a sample widget is:

```
Window XtCreateWidget(parent, arglist, argcount)
Window parent;
ArgList arglist;
int argcount;
```

A widget is capable of both creating a widget instance in a new window and creating a new widget instance in an existing window that is passed in by the application. To accomplish this, all widgets are capable of accepting a window ID as an optional parameter within the `argList` structure. When passed an existing window ID, a widget is permitted to discard the window contents but not permitted, unless directed by the contents of `argList`, to change any other window property.

2.2. Using Widgets

Widgets serve as one of the primary building blocks of a user interface or application environment. The widget set consists of primitive widgets (for example, a command button) and composite widgets (for example, a radio button). As such, these components serve as a default interface for application programmers who do not want to implement their own application interface. In addition, they serve as examples or a starting point for those widget programmers who, using the set of intrinsic mechanisms, want to implement alternative application interfaces.

Once a widget has been created, the application can interact with it using messages. The message interface was chosen so as to allow generic commands to be sent to widgets without concern for the type of the widget. When an application sends a message to a widget, a general toolkit message handling routine first converts the message into a synthesized toolkit X event and then dispatches it to the widget. This enables a widget to be written with a single event entry point through which it receives both X events and client messages.

The generic message types that all widgets are required to support are:

<code>XtAsetvalue</code>	Allows a client to set a number of widget attributes
<code>XtAgetvalue</code>	Allows a client to query a number of widget attributes
<code>XtAredraw</code>	Request widget to redisplay itself
<code>XtAfree</code>	Destroy widget

In addition, widgets are free to define widget specific messages.

2.3. Event Processing

Applications typically create all their widgets and then loop picking up X events and dispatching them. Each widget must provide a single event handling routine that it makes known to the X Toolkit's event dispatcher. Then, when an event is received for this particular widget instance, the event dispatcher calls the widget's event handler passing it the event to be processed.

While processing an X Event, a widget can inform the application by means of a callback facility that some change has taken place. All widgets allow an application to provide at least one callback procedure. If more are needed, a widget can allow more than one callback procedure. For each callback procedure supplied by the application, the application must be allowed to supply a unique tag value. The tag is passed as a parameter when the callback routine is invoked. This provides the means by which the application can determine which widget instance is generating the callback. The syntax for a callback routine is widget dependent.

During the processing of an event, instead of returning to the application and allowing it to dispatch the next event, certain widgets can read another X Event directly. However, because this

goes against the model of allowing the application first crack at an event before it is dispatched, having a widget read events directly is strongly discouraged. The X Toolkit, however, describes a minimum set of rules for such instances. For example, this set of rules states that widgets must push back all events that they read but do not process.

2.4. Widget Composition

Widget composition is the concept of combining several primitive widgets to form a new composite widget. This allows widgets to be recursively grouped together to form new widgets. Because a composite widget is nothing more than a super widget, it provides the same type of programmatic interface to the application as does a primitive widget.

A widget composer takes a group of composite and primitive widgets and blends them to form a new composite widget. A widget composer usually consists of widget-specific code embodied in a complex widget. The composer code is responsible for controlling both the communications between its components and, with the aid of a geometry manager, the relationship between the locations of these components. Some widget composers are fairly rigid in respect to what primitive widgets they are capable of controlling. Other widget composers, such as a panel widget, are more flexible.

3. Intrinsics

The X Toolkit provides a number of intrinsic functions that facilitate widget construction and composition.

3.1. Gathering Input

To process input within the X Toolkit, an application uses one processing control loop that reads, preprocesses, and dispatches input. The functions necessary to read input from the X Toolkit. provide capabilities similar to those provided in X.

The input gathering functions provide mechanisms for normal X input reading (analogous to Xlib), adding or removing input sources other than X to the gathering facility, setting, querying or clearing of timeouts for regular continuous operations (for example, smooth incremental scrolling).

3.2. Dispatching Events

The X Toolkit provides an event dispatcher that can be used in widget event processing. An event handling routine for events pertaining to a specified window may be registered with the dispatcher. Then, when the application calls the routine to dispatch events with an X event, it uses the window field in the event structure to determine the appropriate event handling procedure. The event is then handed to this procedure. While registering an event handler, it is possible to specify the set events one is interested in fielding.

The dispatcher also allows a client to register a notification procedure that is to be invoked when a specified set of events occur, regardless of which window they pertain to. This allows for multiple procedures to process the event before the event is dispatched in the normal manner. A common use of such notification is to get control to destroy a menu created on the button-down transition when a button-up transition occurs in any context.

3.3. Geometry Management

Widgets are not in control of their size and location. The size and location of a widget are controlled by an ancestor of that widget. However, the widget often has the best idea of its optimal size and can also have preferred locations. Geometry management is the mechanism that widgets use to request changes to their size or location.

Generally, each widget has a geometry manager associated with it. Mechanisms are provided for a widget to make a request to its geometry manager. The geometry manager then decides whether to allow the request, disallow the request or suggest a compromise. If it allows the request, the geometry manager then makes the Xlib calls to modify the widget's window.

Although geometry requests are generally made by the widget itself, they can also be made by some external agent. The geometry manager, itself, defines the layout of a group of widgets. Geometry managers can be written to stack widgets in rows and columns, center widgets in windows or arrange widgets in some other fashion.

When a widget makes a request to change its characteristics, the geometry manager, when possible, also rearranges and resizes the other widgets it controls accordingly. These widgets then are informed of changes to them through the usual X event mechanism.

Often, geometry managers find that they can satisfy a request only if they can resize a widget that they are not in control of. In this case, the geometry manager makes a request to that widget's geometry manager. Geometry requests can be nested this way to any depth.

The X Toolkit provides geometry management functions that let you associate and disassociate a geometry manager procedure with a widget, get a geometry manager procedure, or send a geometry change request to a widget's geometry manager.

The types of possible geometry requests are widget resize, widget reposition or restacking a widget above or below all its siblings. It is possible to add new types of geometry requests should the need arise.

If the geometry manager request succeeds, it returns `requestYes`, and the geometry manager changes the widget's geometry. If the geometry manager decides that the request cannot be satisfied, then it returns `requestNo`. If the specified geometry cannot be used and this geometry manager request fails, but the geometry manager is able to specify an alternative geometry, then it returns `requestAlmost` together with the alternative rectangle geometry.

When `requestAlmost` is returned, the widget must decide if this compromise, suggested in the `replyBox`, is acceptable. If the compromise is acceptable, the widget must not change its geometry on its own, rather it should make another request to its manager passing in the compromise geometry.

It is important that widgets cooperate with the geometry management mechanism. Whenever they need to change size or location, they should do it through the geometry manager and be prepared to deal with a different size if the manager refuses to honor their request. This mechanism puts the geometry manager in charge. Were widgets instead in control of their own geometry, there would be no mechanism to arbitrate conflicting geometry changes made by a set of widgets.

3.4. Selection Management

X Version 11 provides a very general mechanism for applications or widgets to communicate with each other by using the server as an intermediary. In general, this mechanism uses "notifications" to signal other windows and "properties" that are stored on the server to pass values. For further information, see those sections on Notification and Selections in the X Version 11 Xlib documentation.

Many user interfaces support the concept of a global or secondary selection. Fundamentally, a selection has the semantics that an “owner” advertises that he is willing to supply the value of the selection in a number of forms (for example, a string, file, or number). A selection of a given type also has the property that there is only one owner that can be active at any point in time. Typically, when a user makes a new selection in the same or another application, the old selection is unhighlighted, ownership is passed to the new selected item, and the new selection is highlighted. To implement this user interface behavior, applications (or widgets) must have a mechanism for requesting and relinquishing ownership of a selection of a specific type.

For these purposes, Xlib Version 11 provides functions for setting a selection and converting a selection. The first makes the requestor the owner of a specific selection, while the second converts the selection into a specified type.

X Version 11 defines three XEvents: SelectionClear, SelectionRequest and SelectionNotify. The first clears (removes) the owner of a selection. The second makes the requestor the owner of the selection and converts it to type request. The third stores the requested selection in a property.

The ConvertSelection function causes a SelectionRequest XEvent to be sent to the owner of the selection. This request contains a type Atom that allows an open ended set of conversions. However, the server does nothing to enforce this, and specific semantics are implied by convention. The property Atom is where the requestor wants the owner to put the returned value.

The following is an example of how a debugger might find out the position in a source file in order to place a source breakpoint.

requestor:	asks for the value of the selection as a “FileName”
owner:	“/udir/karlon/hacks/selection.c”
requestor:	asks for the value of the selection as a “LineNumber”
owner:	93.

Given the file name and the position in the file, the debugger then can set the break point.

Another more common example exists in the implementation of a cut buffer function. To implement this function, an application or widget performs the cut operation and saves the data. It then sets itself as the owner of the cut buffer selection. When anyone wishes to use the contents of the cut buffer, they call the ConvertSelection function and ask for the contents of the cut buffer selection as an ASCII string. This causes a SelectionRequest XEvent to be dispatched to the owner of the cut buffer text. The owner then stores the data as an X property on the requestor’s window and sends a SelectionNotify XEvent back to the requestor.

Because X applications that use a display can be running on a number of different machines, the selection ownership arbitration must be done by the server itself (the only common element). X Version 11 provides functions that are sufficient to perform this task.

3.5. Atom Management

The X Toolkit avoids defining a closed sets of constants to specify options. Ideally, it uses a string for each constant. If it is decided that a new option is needed, it is easy to define a new string. No programs or header files need be modified.

Not only are strings difficult to manage, but string comparisons also are slow. Therefore, the X Toolkit implements an atom mechanism. Atoms are opaque types that can only be compared with each other for equality. Any given string has exactly one atom associated with it. The X Toolkit provides functions to convert between strings and atoms. These functions are similar to those in the Xlib Version 11, but to avoid unnecessary dialog with the server, atoms are implemented locally in the X client.

The X Toolkit uses atoms for specifying options to widgets, for message IDs, as type fields in the context mechanism, as well as for many other purposes.

The atom management functions let you define atoms, retrieve strings from atoms, create unique atoms and initialize atoms.

3.6. Context Management

The context manager provides a way of associating various types of data with a widget. The context manager requires knowledge of the widget ID and the type of the data to store or retrieve data.

The context manager can be viewed as a two-dimensional, sparse array. One dimension is subscripted by the window ID and the other by a type field (atom). Each entry in the array contains a pointer to the data. The X Toolkit provides context management functions with which you can save, retrieve or delete data in this sparse array.

Because widgets are named by their window IDs, widgets rely heavily on the context management facility to map window IDs in client widget requests to the data structure associated with the widget instance corresponding to that window.

3.7. Resource Manager

Widget writers need a consistent, easy-to-use mechanism for finding out what resources (for example, color, font, or border width) to use when creating widgets. The resource manager is a database manager that is specifically tailored to the needs of widget creation.

Applications need a way to provide meaningful application-specific default values and assign explicit values to those resources that are not under user control. They also need some mechanism for discovering and integrating global defaults and user preferences into the defaults that they provide to their widgets.

Users need a flexible method for specifying preferences for various resources. A system that both specifies defaults and refines them not only seems to provide the necessary flexibility but also seems to be easy to understand and use. For example, the user might normally want text to be in a sans-serif 10 point font. But, in a mail tool, the user might prefer a serif font and would like its command buttons to be bold. It should be possible to specify these preferences in a natural, easy-to-use manner.

Consider an X-based mail reading application called `xmail`. At the top level, it might consist of a paned window. One pane of the paned window is a button box widget of command buttons, named `toc`. One of these command buttons is used to include (fetch) new mail. This widget has a name `"xmail.toc.buttons.include"` and a class `"application.panelwindow.buttonbox.commandButton"`. Its name is the name of its parent, `"xmail.toc.buttons"`, followed by its name `"include"`. Its class is the class of its parent, `"application.panelwindow.buttonbox"`, followed by its particular class, `"commandButton"`. The fully qualified name of an attribute is its name appended to the widget name, and its class is its class appended to the widget class.

This button needs the following resources: title string, font, foreground color, background color, foreground color for its active state, background color for its active state. Each of the resources are considered to be attributes of the button and, as such, have a name and a class. For example, the foreground color for the button in its active state might be named `"activeForeground"` and its class would be `"color"`.

When a widget requests a resource (for example, a color), it passes the complete name and class of the resource along with the desired representation type to a lookup routine. The representation type lets a widget request different representations for the same resource. For example, a color

might be requested as a color record, a pixel, a pixmap, or a name string. Rather than require the application to store every possible representation of a resource, the X Toolkit provides a mechanism for converting between representations.

The widget interface comes in two layers. The top layer allows applications to store resources by name, class, and representation type, and allows applications to retrieve them given a fully qualified name, class, and destination representation. The resource manager automatically calls a conversion routine, if necessary and possible, to convert the stored representation to the destination representation.

This layer is built on top of a primitive manager that provides the ability to store entries by name and class and a way of retrieving these values given a full name and class. This layer stores uninterpreted variable length values and has no knowledge of resource representations.

The algorithm for determining which resource name or names match a given query is the heart of the database. The idea is that resources are stored with only partially specified name and classes. The unspecified portions of the name match any part of a more completely specified name or class. In particular, all queries fully specify the name and class of the resource needed. The lookup algorithm then searches the database for the name that most closely matches this full name and class.

The definition of a match is as follows:

For a query of name $N = n1.n2.n3...nk$ and class $C = c1.c2.c3...ck$, a partial name $P = p1.p2.p3...pm$ matches (N,C) , if P matches the regular expression $[n1lc1] [n2lc2] [n3lc3]...[nklck]$. The regular expression "alb" matches either "a" or "b", "[a]" matches "a" or NULL (that is, "a" is optional), and "a b" matches "a" followed by "b". As they are defined, the name and the class have exactly the same number of components.

For two partial names $P1$ and $P2$ that both match (N,C) , the definition of the one that "most closely matches" is still an open issue.

For example, assume the following user preference specification:

xmail.background:	red
button.font:	Helv10
button.background:	blue
button.color:	green
xmail.toc.button.activeForeground:	black
xmail.toc.buttons.border:	3

A query for the name "xmail.toc.buttons.include.activeForeground" and class "application.panelwindow.buttonbox.button.color" matches "xmail.toc.button.activeForeground" and returns "black". However, it also matches "button.color". The "xmail.toc.button.activeForeground" specification is clearly the correct one.

The type conversion machinery calls conversion procedures to convert between differing resource representations. There are some predefined conversions, but clients can register as many new conversions as are needed. These registered conversion procedures take a source type and value and convert them to a destination type and value. There is an atom for each defined resource representation type, and the values are size and address pairs.

The X Toolkit provides resource management functions that let you store and get values, store and get resources, convert values, retrieve and store databases and configure resource databases.

The representation of a resource database in some non-volatile form has not been specified. Presumably, each representation type will have a procedure to turn the internal representation into

a canonical non-volatile representation and back again. Both how these procedures are specified and how they get called have not been defined.

The order in which partial names are matched with full names and classes is not well defined. For simple cases, the definition is clear. You would like partial names with longer name prefixes to match before those with shorter name prefixes. For example, you would like “xmail.toc.buttons.color” to match a request for “xmail.toc.buttons.include.foreground” and “application.panelwindow.buttonbox.button.color” before “xmail.toc.buttonbox.button.color”. Buttons is the NAME of a particular instance of a toc buttonbox and, therefore, is more specific. The ordering is not so clear, however, when a partial name contains a mixture of name and class atoms and not a prefix of name atoms with no gaps, followed by an arbitrary selection of class atoms.

In general, you should only store partial names with contiguous name prefixes (possibly empty) followed by an arbitrary class definition. The ordering in that case is:

1. A string with the longer name prefix
2. A string with the "more specific" class definition.

For example, the order of class definitions matches for a complete class definition of “mailreaders.panelwindow.buttonbox.button.color” is:

```
mailreaders.panelwindow.buttonbox.button.color
mailreaders.panelwindow.buttonbox.color
mailreaders.panelwindow.button.color
mailreaders.panelwindow.color
mailreaders.buttonbox.button.color
mailreaders.buttonbox.color
mailreaders.button.color
mailreaders.color
panelwindow.buttonbox.button.color
panelwindow.buttonbox.color
panelwindow.button.color
panelwindow.color
buttonbox.button.color
buttonbox.color
button.color
color
```

Many issues of subclassing of widgets can be raised. For example, **include** is really a command button which can reasonably be considered a subclass “button”. While naming **include** a “commandButton” solves this particular problem, it does not solve the subclassing problem in general. On the other hand, unless the X Toolkit provides some way of adding functionality to an existing widget without creating a complete new widget class, it really does not have a subclassing problem. Without inheriting semantics from anywhere else, each widget is entire unto itself. Strictly speaking, there is no widget subclassing.

The method by which a widget determines its name and class is not specified in this interface. One method is to put the specific name and class of a widget as attributes on its window. Another is to pass the parents name and class to each subwidget as part of the creation data. In either case, this problem should not pose any serious difficulties.

3.8. Translation Management

The translation manager provides an interface to specify and manage the mapping of X Event sequences into widget supplied functionality. The simplest example would be to call procedure "foo" when key "y" is pressed. This interface provides users the ability to specify or customize event bindings. It also provides clients the ability to export the functions they implement and to provide default bindings to those functions.

The translation manager uses two tables to perform translations. Usually, the user, by means of the resource manager, supplies a set of event-sequence-to-function-name bindings. Then, the widget supplies a set of function-name-to-function-implementation bindings. The translation manager's job is to match the user's intentions into the widget's exported functions. To use the translation manager, a widget performs the following steps. Either get the user's event bindings from the resource manager, or supply defaults. Set bindings by calling **XtSetActionBindings** and pass both the event bindings and widget specified action bindings as arguments. Call **XtTranslateEvent** for each X Event to be translated. The result of the translation is returned as a list of action tokens. The interpretation of the tokens is widget implementation specific.

Mapping the X Event to a function name is accomplished by specifying an event sequence binding. Event sequence bindings exist in textual and compiled forms. The compiled form is private to the translation manager. The textual form has the following syntax:

Mode<EventType>Detail: {atom | "string" | 'char' }

The information on the left specifies the sequence of X Events, while that on the right specifies what to do when that sequence is detected. The interpretation of the information on the right is specific to the widgets. A common use is for atoms to name functions and for strings, chars, and numbers to mean "self insert" (that is, act as though this was just typed).

The Mode field is used to specify normal X keyboard and button modifier mask bits (Control, Shift, Meta, Lock). The EventType field describes XEvent types. The currently defined EventType values include KeyPressed and KeyReleased (with and without keyboard modifiers), ButtonPressed, ButtonReleased, and MouseMoved. The Detail field is event specific and normally corresponds to the detail field of an X Event, for example, <Key>A. In the event that no event bindings are specified externally, the widget specifies the default bindings in textual form.

Clients of the translation manager must provide a table (array) of action names to function implementation bindings using the following typedef.

```
typedef struct _XtActionsRec {
    char *string;
    caddr_t value;
} XtActionsRec, *XtActionsPtr;
```

The string field is converted to an atom and is interpreted as the name of the action function. The value field is a pointer to any client supplied data. A common use of this field is to supply a procedure to call for the named function.

The X Toolkit provides translation management functions that let you compile default event bindings, set and get action bindings and translate an X event.

This design provides functionality that is basically optional. This means that widget writers are free to use all, part, or none of these facilities. It also provides this functionality in a number of separate steps. This allows the client considerable control structure flexibility in how translations are done.

By defining the calling sequence to all widget-supplied functions, the two-step translation process could be made into a one-step process. This will allow the translation manager to call widget

functions directly.

Any general solution of the translation problem involves defining a finite state automata and a language for specifying state transitions. This, in turn, implies implementation of a compiler and/or interpreter for this language. By solving the event translation problem in this manner, the specification of simple and straightforward uses (for example, Control-H is delete character) becomes difficult and results in considerable design and implementation effort that is used infrequently.

3.9. Error Handling

The X Toolkit provides an error handling interface. This facility allows a client to register a procedure to be called whenever an error occurs. This facility is intended for error logging but neither for error correction or recovery. It is modelled after the Xlib error functionality. The error handler can be changed to a user supplied routine.

Along with the error description, the global variable `XtreferenceCount` is printed out by the default error handler. `XtreferenceCount` is the number of toolkit calls made when the error handler was invoked, since the start of the application.

4. Sample Widgets

The sample widget set to be provided with the X Toolkit consists of:

Command Button	A widget that may be "pressed" by a user to notify an application.
Boolean Button	Similar to a Command Button, except that a boolean state is displayed to the user. The state toggles each time the button is pressed.
Valuator	An analog input widget.
Label	A widget that is output only (insensitive to the user).
Text Subwindow	Provides an application a means of displaying multiple lines of text. The user may edit this as well as select pieces of text.
Numeric Subwindow	A widget used for digital numeric input.
Titlebar	A label together with some command buttons.
Scrollbar	Similar to a valuator. Normally used to give user scrolling control over another window or widget. Differs from a valuator in that the user is given feedback as to what fraction of the "scroll" is currently visible.
Radio Button	A one-of-many selection widget.
Checkbox	A n-of-many selection widget.
Text Edit	A widget for entering and editing a line of text.
Raster Select	A one-of-many selection widget that uses pictures.
Cascading Menu	A pop-up style menu tree that a user can navigate.
Dialog Box	A widget consisting of a label, an editable text field and some command buttons.

5. Acknowledgements

Our thanks go to the many people who have contributed to the design of the X Toolkit. These include Tom Benson, Mike Gancarz, Charles Haynes, Harry Hersh, Phil Karlton, Kathy Langone, Mary Larson, Joel McCormack, Leo Treggiari, Jake VanNoy, Terry Weissman, (all from DEC); Phil Gust, Frank Hall, Tom Houser, Ed Lee, Rick McKay, Jack Palevich, Fred Taft, Ted Wilson, (all from HP); Ron Newman and Ralph Swick (from MIT-Athena).

6. References

The X Toolkit — A Proposed Architecture, December 17, 1986, MIT X.V10R4 distribution.

Jim Gettys, Ron Newman, Bob Scheifler, *Xlib — C Language X Interface, Protocol Version 11*, Alpha Update, April 1987, MIT.

The X Window System Protocol Version 11, Alpha Update, April 1987, MIT.

Shared Libraries in SunOS

Robert A. Gingell

Meng Lee

Xuong T. Dang

Mary S. Weeks

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043

ABSTRACT

The design and implementation of a shared libraries facility for Sun's implementation of the UNIX[†] operating system (SunOS) is described. Shared libraries extend the resource utilization benefits obtained from sharing code between processes running the same program to processes running different programs by sharing the libraries common to them.

In this design, shared libraries are viewed as the result of the application of several more basic system mechanisms, specifically

- kernel-supplied facilities for file-mapping and "copy-on-write" sharing;
- a revised link editor supporting dynamic binding; and
- compiler and assembler changes to generate position-independent code.

The use of these mechanisms is transparent to applications code and build procedures, and also to library source code written in higher-level languages. Details of the use and operation of the mechanism are provided, together with the policies by which they are applied to create a system with shared libraries. Early experiences and future plans are summarized.

1. Introduction

The UNIX operating system has long achieved efficiencies in memory utilization through sharing a single physical copy of the *text* (code) of a given program among all processes that execute it. However, a program *text* usually contains copies of routines from one or more libraries, and occasionally a program consists *mostly* of library routines. Considering that virtually every program makes use of routines such as *printf*(3), then at any given time there are as many copies of these routines competing for system resources as there are different active programs.

In an environment containing single-user systems, such as workstations, the likelihood of achieving much benefit from sharing multiple copies of entire programs seems small. As the number of programs in a system increases (a guaranteed attribute of each new system release), so does the waste in file storage resources containing yet more copies of common library routines. Thus, there is increasing motivation to extend the benefits of sharing to processes executing *different* programs, by sharing the libraries common to them.

This paper describes the design and implementation of a shared libraries facility for Sun's implementation of the UNIX operating system, SunOS. We discuss our goals for such a facility, our approach to its design and implementation, and our plans for its use. We also discuss our early experiences, and our plans

[†] UNIX is a trademark of AT&T.

for the future.

2. Goals

Most of our goals were driven by a desire to have a facility that was as simple to use and evolve as possible. We also wanted to provide mechanisms that were as flexible as possible, so that the work we performed could be used to support other activities and projects. Providing mechanisms with great apparent simplicity would also help motivate their use. To that end, we arrived at the following specific goals:

- **Minimize kernel support.** Clearly, any support we put in the kernel would be very inflexible, and further complicate an already complex environment. We considered an ideal situation to be one involving no kernel changes.
- **Do not require shared libraries.** Although we might make the use of shared libraries the default system behavior, we felt we could not *require* their use or otherwise build fundamental assumptions requiring them into other system components.
- **Minimize new burdens.** The introduction of any new facility creates the potential for new burdens to be imposed upon its users. To minimize these, we decided how shared libraries should impact various groups:
 - **Application programmers:** The use of shared libraries must be transparent to application source code, program build procedures, and the use of standard utilities such as debuggers. It was also considered desirable to be able to use existing object files.
 - **Library programmers:** That a body of library code is to be built as a shared library must also be transparent to its source code. However, it need not be transparent to the procedures used to build the library, and such a goal appeared contradictory in any case – someone has to decide that a shared library will be built. The goal to not change library source was a direct consequence of not having the resources to change the large amount of library code already in existence. Even source alterations such as those used with System V shared libraries [ARNO 86] appeared more than we wished to do.
 - **Administrative:** There should be no requirement to administer and coordinate the allocation of address space. Libraries should be able to evolve and be updated without requiring rebuilding of the programs that used them as long as their interfaces are compatible, and mechanisms would have to be available to handle interface changes.
- **Improve the environment.** Where possible, we wanted our changes to provide functional benefits beyond the resource utilization ones we expected. This included having a great deal of flexibility in easily testing updates to libraries.
- **Performance.** Shared libraries represents a classic time vs. space trade-off opportunity. We were deferring the work of incorporating library code into an address space in order to save both secondary and primary storage space. Thus, we expected to pay a time penalty in programs using shared libraries. However, the expectation was that if sharing of library code really occurred, then the I/O (real) time required to bring in a program and get it executing would be greatly reduced. As long as the CPU time required to merge the program and its libraries did not exceed the I/O time we saved, the apparent performance would be the same or potentially even better. This approach fails if sharing does not occur, or if the system is CPU saturated already.

Even though a moderate cut in I/O time offers a large window for computation, we felt that an attempt to equal the performance of current systems was unrealistic, and instead set two performance goals permitting a limited degradation in CPU performance for programs that used shared libraries. These goals were:

- $\leq 10\%$ for programs not dominated by start-up costs; and
- $\leq 50\%$ for programs that were dominated by start-up costs.

A program was considered to be dominated by start-up costs if it took less than half a second to execute on a Sun-3/75.

3. Approach

Given our goals for flexibility, the most productive approach was not to build a mechanism *specific* to shared libraries. Rather, by abstracting the general properties we required of shared libraries and providing mechanisms to deliver those properties directly, we hoped to achieve the sought-for benefits and flexibility to address the needs of other projects. The mechanisms we chose were:

- a high degree of memory sharing of general objects (e.g., files) at a fine level of granularity (pages);
- a revised system link editor (*ld*) that supports dynamic loading and binding; and
- compiler changes to generate *position-independent* code (PIC) that need not be relocated for use in different address space arrangements and thus may be directly shared.

3.1. Memory Sharing

The mechanism that provides our memory sharing is a new Virtual Memory (VM) system for SunOS. Although more completely described elsewhere [GING 87], the principal features of the new system include:

- file mapping as its principal mechanism, accessed by programs through the *mmap(2)* system call;
- sharing at the granularity of a file page; and
- a per-page copy-on-write facility to allow run-time modification of a shared object without affecting other users of the object.

The new VM system uses these features internally, so that the act of *exec*'ing a program is reduced to the establishment of copy-on-write mappings to the file containing the program. A "shared library" is added to the address space in exactly the same way, using the general file mapping mechanism. The use of files in this way originated with MULTICS [ORGA 72], and the use of file page mapping to incorporate library support at execution time was established with TENEX [MURP 72] and its evolution as Digital Equipment's TOPS-20. Comparable approaches have been applied with UNIX-based systems as described in [SZNY 86] and [DOWN 84].

3.2. New *ld*

The changes to *ld* reflect an observation that the activities that must occur to execute a program with shared libraries are no different than those to execute one without them, at least conceptually. All that has really changed is when, and over what scope of material, those activities occur. Conceptually, *ld* has been turned into a more general facility available at various times in the life of a program (in perhaps different guises) to perform its link editing function.

The old *ld* built all programs *statically*. Executable (*a.out*) files contained complete programs, including copies of necessary library routines. Executables were created by link editing the program in (usually) a single batch operation using *ld*. *ld* would refuse to build an incomplete executable file.

The new *ld* will build "incomplete" *a.out* files, deferring the incorporation of certain object files until some later time (generally program execution). These deferred link editing operations employ the system's memory management facilities to map to and thus share these objects directly. A "shared library" is simply the code and data constituting a library built as such a *shared object* (*.so*) file. A *.so* is simply one of these "incomplete" *a.out* files that lacks an entry point. It should be noted that a *.so* file can be *any* object, a "library" is simply one of many possible semantic uses for it.

As previously noted, dynamic link editing is still essentially the same operation as static link editing, but occurring at a different time. A link editing operation effects some change to either the material being added, or that to which it is added, or more likely both. However, when an object is changed as the result of such processing, it can no longer be shared with other users of the object, as the change is unlikely to be useful to any program other than the one in which it occurs. Such changes are accommodated automatically by the VM system, using its copy-on-write facilities to create per-process private copies of the pages of the file the process attempts to modify. Thus, the extent of the changes a link edit performs affects the degree to which sharing can occur.

Although a dynamic link edit operation may impact the degree to which sharing can occur in a system, it does not affect the *correctness* of the resulting program. A strong characteristic of our approach is this separation between “right and wrong” vs. “good and bad”. Almost any legitimate combination of objects can be link edited into a program at any time (e.g., there are very few “wrong” combinations), but those that maximize sharing will be “best”.

3.3. PIC

In the previous section it was observed that code that minimizes the amount of dynamic link editing promotes sharing and is thus “best”. To increase the prospects for having the “best” code, we changed our C compiler to optionally generate *position-independent* code (PIC). PIC needs link editing only to relocate references to objects external to the body of code that has been built as PIC, and is thus more shareable. Again, it is not *necessary* to have PIC, just *better*.

However, PIC programs will be slower than non-PIC ones. To localize the link editing for references to global objects, the code refers to such objects indirectly through *linkage tables*. The specific amount of degradation is a function of the number of dynamic references to global objects.

4. Mechanisms

The previous section provided an overview of the approach we have employed, and briefly identified the mechanisms we would use. With this background, we describe the mechanisms in greater detail.

4.1. Compiler Changes

The C compiler has been altered to take a new option (`-pic`) that causes it to generate PIC. When `-pic` is specified, the code generated by the compiler changes in the following ways:

- Each function prologue is extended to include the initialization of a register that is used as the base address of a *linkage table* to global objects, this table is called the *global offset table* (*GOT*). For the Motorola 680x0 used in Sun’s workstations, this code is:

```
movl #__GLOBAL_OFFSET_TABLE, a5      | Get offset to GOT
lea pc@(0, a5:L), a5                 | Get absolute address
```

which computes the absolute address of the *GOT* associated with this function based on a PC-relative offset from the function prologue to the table. The register `a5` is unavailable for the life of the function, and is one of those that the compiler expects called functions to preserve.

- Each reference to a global data object is generated as a dereference of a pointer in the *GOT*. For example, a reference to the external integer `errno` in C is generated as:

```
movl a5@(_errno:w), a0               | Get address of _errno
movl a0@d0                           | Get contents
```

Currently, the code generation scheme for static data objects is identical to that used for globals. This represents an area for future optimization work.

- Each function call is generated as an assembler pseudo-operation including a “free register”, for example:

```
jbsr _foo, a0                        | _foo()
```

for an expression involving a call to the function `foo`. The assembler will, if `_foo` is undefined to it, expand the pseudo-operation to an instruction sequence that involves loading a PC-relative reference to an entry in a *procedure linkage table* into the “free register”, and then issuing a subroutine call instruction involving the PC in the calculation of the effective address.

The code sequences generated for the 680x0 assume that the linkage tables are of a limited size, specifically no larger than 64K bytes. In the event the tables require a larger size, the compiler can be coerced into generating more clumsy code sequences permitting linkage tables to a full 32 bits in size (by expressing `-pic` as `-PIC`). However, we have yet to find a program that requires the use of this option.

4.2. Assembler Changes

The code generated by the compiler with the `-pic` option requires support from the assembler. The support required is that the assembler generate some new relocation information for certain constructs, and a change in interpretation for some syntactic forms. This support is enabled by the assembler flag `-k`, and is generated automatically by the C compiler driver when invoking the assembler for a compilation that contained the `-pic` or `-PIC` options.

When assembling a module with the `-k` flag enabled, the assembler:

- interprets a relocatable expression in an operand involving an “immediate” addressing mode as a PC-relative reference to any symbol involved and generates a PC-relative relocation record for the expression;
- interprets symbolic relocatable expressions in operands involving base-register relative addressing as a reference to the *GOT* entry for the symbol and generates a relocation record indicating such; and
- generates a “procedure call” relocation type for all `jsr` pseudo-operations it assembles.

It should be noted that although examples have been provided using an assembler for the 680x0 processor employed in Sun workstations, the requirements for these special relocation types are architecture independent.

4.3. Link Editor changes

The most extensive changes have been performed to the link editor, *ld*. These not only include changes to the batch form of the link editor (embodied as *ld*), but also the creation of an execution-time version (*ld.so*).

4.3.1. Batch link editor (*ld*)

The batch link editor, *ld*, combines a variety of module types to produce an *a.out* file. How that *a.out* file can be used is very much dependent on what *ld* can determine to do with the information it has been fed. Whereas the previous version of *ld* had to determine *everything* about a program, the new version simply stops working when it runs out of information on the assumption that later events will provide more.

ld's output can be one of two basic types, including:

- a “simple object” (*.o* file), produced by simply combining other *.o*'s into a single, larger one (`-r` flag);
- an “executable” (*a.out*), which is either a “program” (has an entry point) or a *shared object* (*.so*) (does not have an entry point).

The production of a *.o* file through the use of the `-r` flag is a special use of *ld* that, while useful, is not relevant to the issues being discussed and will not be considered further.

Exactly what gets produced depends on what *ld* was fed in the way of input files and command line options. *ld* will process the following kinds of input files:

- simple object files, *.o*'s;
- *archives*, *.a*'s, conglomerates of simple objects and also referred to as *libraries*; and
- *shared objects*, *.so*'s, also known as dynamically bound executables and sometimes called *shared libraries*.

Each *.o* file is simply concatenated to previous *.o* files in the order it is encountered. In this respect, *ld* is unchanged except that it handles the new relocation operations required by code the assembler generated as PIC.

Each *.a* is searched exactly once as it is encountered – only those entries matching an unresolved external reference are extracted and concatenated. Again, this is exactly as *ld* has always done, with the addition of PIC handling.

Any *.so* encountered is (usually) searched for symbol definitions and references, but does not contribute any material to be concatenated except under certain conditions involving other options (described further below). However, their occurrence in the command line is stored in the resulting *a.out* file and utilized by the execution-time *ld.so* to effect dynamic loading and binding.

ld's **-l** flag is used to specify a short name for an object file to be used as a library. The full name of the object file is derived by adding the prefix *lib* and a suffix of either *.a* or *.so* (for archive or shared library, respectively). The specific suffix applied depends on the binding "mode" *ld* is operating in at the time the **-l** flag is processed. *ld*'s binding "mode" is specified by a new flag, **-B** that takes several keyword arguments:

dynamic	Allow dynamic binding, do not resolve symbolic references, and allow creation of execution-time symbol and relocation information. This is the default setting.
static	Force static binding, implied by options that generate non-sharable executable formats.

-Bdynamic and **-Bstatic** may be specified multiple times and may be used to toggle each other on and off. Like **-l**, their influence is dependent upon their location. When **-Bdynamic** is in effect, any **-l** searches may be satisfied by the first occurrence of either form of library (*.so* or *.a*), but if both are encountered the *.so* form is preferred. Since **-Bdynamic** is the default setting, the use of shared libraries in the construction of a program thus "falls out" from simply installing the *.so* that represents the shared library in the library search path used by *ld*.

If **-Bstatic** is in effect, however, *ld* will refuse to use any *.so* forms of libraries it encounters and continue searching for the *.a* form. Further, an explicit request to load a *.so* file is treated as an error.

After *ld* has processed all its input files and command line options, the form of the output it produces is based on the information it has been able to discern. *ld* first tries to reduce all symbolic references to relative numerical offsets within the executable it is building. To perform this "symbolic reduction", *ld* must know that either

- all information relating to the program has been provided, in particular, no *.so* will be added at execution time; and/or
- this program has an entry point and symbolic reduction can be performed for all symbols having definitions existing in the material it has been provided.

It should be noted that uninitialized "common" areas (essentially all uninitialized C globals) are allocated by the link editor *after* it has collected all references. In particular, this allocation can not occur in a program that still requires the addition of information contained in a *.so* file, as the missing information may affect the allocation process. Initialized "commons", however, are allocated in the executable in which their definition appears.

After *ld* has performed all the symbolic reductions it can, it attempts to transform all relative references to absolute addresses. *ld* is able to do this "relative reduction" only if it has been provided *some* absolute address, either implicitly through the specification of an entry point, or explicitly through other *ld* options. If, after performing all reductions it can, there are no further relocations or definitions to perform, then *ld* has produced a completely linked executable – essentially its old behavior.

However, if any reductions remain, then the executable being produced will require further link editing at execution time in order to be useable. In the data spaces of such executables, *ld* creates an instance of a *link_dynamic* structure that has the label `__DYNAMIC`. The *link_dynamic* structure has the form:

```
struct link_dynamic {
    int      ld_version;           /* Version # */
    struct   link_map *ld_loaded; /* Loaded objects */
    long     ld_need;             /* Needed objects */
    long     ld_got;              /* Global offset table */
    long     ld_plt;              /* Procedure linkage table */
    long     ld_rel;              /* Relocation table */
    long     ld_hash;             /* Symbol hash table */
}
```

```

        long    ld_stab;                /* Symbol table itself */
        long    (*ld_stab_hash) ();    /* Hash function */
        long    ld_buckets;            /* Number of hash buckets */
        long    ld_symbols;            /* Symbol strings */
        long    ld_text;                /* Size of text area */
};

```

This data structure is used by *ld.so* to obtain *.so*'s on which this executable depends, and to find the symbolic and relative reduction operations that remain to be performed. The *link_dynamic* structure contains elements that allow evolution of the interfaces to occur without invalidating existing programs. These include the *ld_version* element, and the incorporation of the hash function for the execution-time symbol table as part of the executable.

4.3.2. Relocation of PIC

As described previously, code generated as PIC contains several new relocation record entries: PC relative, references to entries in a *global offset table (GOT)*, and references to entries in a *procedure linkage table (PLT)*.

PC relative relocations are easily handled by *ld*: the value replacing the relocation is simply the offset between the location reference and definition of its target.

GOT and *PLT* entry references are more complex, however. Both of these data structures are allocated by *ld* as part of creating an executable comprised of at least one PIC module, that is, a module containing either *GOT* or *PLT* or both relocation forms. *ld* is responsible for assigning entries in each of these tables for each unique symbol referenced in either a *GOT* or *PLT* reference, and creating a new relocation entry for the table entry. The resulting relocations are then processed just like any other handled by *ld*, by first attempting symbolic and then relative reductions. The table entries themselves are (at least conceptually) indirect pointers to the targets of global references.

4.3.3. *crt0*

Every main program produced by the standard languages is linked with a program prologue module, *crt0*. This module actually contains the program's entry point, and performs various initializations of the environment prior to calling the program's main function or logical starting point. *crt0* was modified to contain a reference to the symbol *__DYNAMIC*. As described above, when *ld* builds an executable requiring execution-time link editing, it defines this symbol as the address of a data structure containing information needed for execution-time link editing operations. If the structure is not needed, any reference to the symbol *__DYNAMIC* is relocated to zero.

Thus, at program start-up, *crt0* tests to see whether or not the program being executed requires further link editing. If not, *crt0* simply proceeds with the execution of the program as it always has – no further processing is involved. However, if *__DYNAMIC* is defined, *crt0* opens the file */lib/ld.so* and requests the system to map it into the program's address space via the *mmap* system call. It then calls *ld.so*, passing as an argument the address of its program's *__DYNAMIC* structure. *crt0* assumes that *ld.so*'s entry point is the first location in its text. When the call to *ld.so* returns, the link editing operations required to begin the program's execution have been completed.

4.3.4. *ld.so*

After *crt0* transfers control to *ld.so*, *ld.so* executes a short bootstrap routine that performs any relocations *ld.so* itself requires. The process of building *ld.so*, described further below, results in only very simple forms of relocation that can be easily handled by this bootstrap routine. *ld.so* then processes the information contained in the *__DYNAMIC* structure of the program that called it, in order to perform the link editing required to start execution of the program.

ld.so's first action is to examine the *ld_need* entry of the program's *__DYNAMIC* structure. This entry contains an offset relative to the *__DYNAMIC* structure of an array of *link_object* structures. Each element of the array has the structure:

```

struct link_object {
    char    *lo_name;           /* Name of object */
    int     lo_library : 1;     /* Library search */
    short   lo_major;          /* Major version */
    short   lo_minor;          /* Minor version */
};

```

and identifies a *.so* that must be added to the program's address space and link edited. The identification is the name specified on the *ld* command line used to build the program, and includes a bit indicating whether the object was named explicitly or via an *ld -l* option. Some version control information is also recorded, however a discussion of the use of this information is deferred.

For each entry in the *ld_need* array, *ld.so* looks up the file identified and maps it into the process's address space. The location in the address space to which the *.so* is mapped is left to the system to decide, and a given *.so* may reside at different locations in the address spaces of different processes. Failure to find a needed object is a fatal error and results in the program's termination. At the end of the initial program's *ld_need* array, *ld.so* examines the `__DYNAMIC` structure of the first *.so* file it mapped in. It processes that *.so's* *ld_need* array, and proceeds likewise through all the loaded *.so's*. Any references to already processed *.so* files are ignored.

For each *.so* that is loaded, *ld.so* builds a *link_map* data structure having the form:

```

struct link_map {
    caddr_t lm_addr;           /* Address mapped */
    char    *lm_name;          /* Absolute pathname */
    struct  link_map *lm_next; /* Next .so */
};

```

Each such structure is placed on a singly linked list in the order it was loaded. The head of the list is rooted in the *ld_loaded* member of the initial program's `__DYNAMIC` structure. This ordering of the loaded *.so's* is used to establish the search order for undefined symbol look-ups.

After all of the modules comprising the complete program have been placed in the address space, *ld.so* attempts to complete the link editing operations begun by *ld*. Specifically, it attempts to perform first symbolic and then relative reductions on all the references *outside of procedure linkage tables* left in the program. In particular, this includes the allocation of any uninitialized commons (since all information regarding their use is finally present). If all non-procedural references can not be reduced to absolute addresses, then it is because a definition for a given symbol is not available, in which case *ld.so* terminates the program with an "undefined symbol" diagnostic.

All non-reduced references in any *PLT's* in the loaded executables are not processed during program startup. Rather, all such references are initialized to cause the initial calls to the procedures they reference to result in the transfer of control to *ld.so*. Upon receiving control from such a reference, *ld.so* will reduce the original reference to the appropriate absolute address and modify the referencing *PLT* entry to direct future calls directly to the targeted procedure. Deferring the binding of procedure entry points until their first reference saves performing perhaps thousands of unnecessary bindings to entry points programs may never call.

4.3.5. Version Management of *.so's*

The previous discussion of the handling of *.so* files in the course of processing an *ld -l* option was simplified with respect to *.so* version control. One of the goals of our project was to accommodate the evolution of shared libraries: to permit them to be updated without impacting the programs that used them so long as the interfaces remained compatible.

The *.so* files used as shared libraries actually employ a more complex name than has been described so far, involving a suffix that describes the version of the library contained in the file. Thus, interface version "2" of the C library, in its third compatible revision, would be placed in a *.so* having the name *libc.so.2.3*. The suffix may actually be an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the operation of the link editors at this time.

The first component is called the library's "major version" number, and the second component its "minor version" number. When *ld* records a *.so* used as a library, it also records these two numbers in the database used by *ld.so* at execution time. When *ld.so* finally searches for libraries, it uses these numbers to decide which of multiple versions of a given library is "best", or whether *any* of the available versions are acceptable. The rules it follows are:

- **Major Versions Identical:** the major version used at execution time must exactly match the version found at *ld*-time. Failure to find an instance of the library with a matching major version will cause a diagnostic to be issued and the program's execution terminated.
- **Highest Minor Version:** in the presence of multiple instances of libraries that match the desired major version, *ld.so* will use the highest minor version it finds. However, if the highest minor version found at execution time is less than the version found at *ld*-time, a warning diagnostic will be issued, although execution will continue.

The semantics of version numbers are such that major version numbers should be changed whenever interfaces are changed. Minor versions should be changed to reflect compatible updates to libraries, and programs will silently prefer the highest compatible version they can obtain. If minor version numbers drop, then although the interfaces should remain compatible, it is possible that certain bug fixes or compatible enhancements that the program builder wanted are unavailable: hence the warning diagnostic.

Although the mechanisms for supporting version evolution of shared libraries have been provided, we have not yet provided any tools to automate their use. As before, the detection of incompatibilities remains the responsibility of the library developer.

4.3.6. Link Editor Environment Variables

ld interprets the values of the environment variables `LD_LIBRARY_PATH` and `LD_OPTIONS`.

`LD_LIBRARY_PATH` augments *ld*'s built-in rules for directories to be used when searching for libraries specified with the `-l` option. If defined, the value of `LD_LIBRARY_PATH` should be a colon-separated list of directory names (as for the `PATH` variable of *sh*). The list specified by `LD_LIBRARY_PATH` is prepended to the list of *ld*'s built-in rules, and follows any further directories specified on the command line with `-L` options.

`LD_OPTIONS` specifies a default set of options to *ld*. `LD_OPTIONS` is interpreted by *ld* just as though its value had been placed on the command line immediately following *ld*'s invocation, as in:

```
% ld $LD_OPTIONS ... other ld arguments ...
```

ld.so also interprets the `LD_LIBRARY_PATH` environment variable, and may be used to substitute test versions of libraries in their own environments at execution time.

4.3.7. Considerations of Dynamically Linked Programs

Beyond providing a basis for improved sharing of system resources, the ability to defer the binding of library and other code offers a number of other potential advantages in terms of increased flexibility for maintenance and development. However, the environment they create is also inherently more complex, something that the policies governing the application of the mechanisms must address. Some aspects of this more complex environment include:

- **Multiple files:** a dynamically bound program consists not only of the executable file that is the output of *ld*, but also of the files referenced during execution. Moving a dynamically bound program *may* also involve moving a number of other files as well. Moving (or deleting) a file on which a dynamically bound program depends may prevent that program from functioning.
- **Ubiquitous link editor:** the previous behavior of *ld* was to produce only a fully linked executable. Link editing issues could be forgotten or ignored once the executable had been successfully produced. However, deferring some of the link editing means (potentially) deferring some of the errors that could occur. With the new facilities, it is possible for a running program to produce a link editor error.

Consider the following example: a programmer misspelling in the use of the function call *printf* results instead in a reference to "pintf". During testing of the code in which the

misspelling occurs, no path to the “printf” reference is ever exercised. However, a later production user does exercise the path. The (no doubt surprised) user will find the program terminated with the message: “_printf: undefined”.

To deal with such problems, *ld* has been provided with an assertion-checking facility that (among other things) can be used to determine if a given program will encounter undefined symbols during execution *if used with the dynamic objects now on the system*. Later erroneous changes to such dynamic objects might still create this problem, however. Program builders wishing to isolate themselves from such problems should simply link their programs statically.

- **Semantic Differences:** there are some semantic differences between the dynamic and static binding algorithms. The differences are not expected to manifest themselves as problems with existing programs, unless such programs engaged in questionable practices in their use of library search ordering. The major semantic difference that can create a problem involves old programs built from several components, where several of those components suddenly become dynamically loadable and others remain static.

Consider the *ld* command:

```
% ld -o x ... <dc> <sc>
```

The executable *x* consists of several objects including a dynamic component (<dc>) and a static component (<sc>). <dc> was, prior to the introduction of the new mechanisms, an unordered archive file. <dc> and <sc> both contain definitions for the symbol *bar*. In addition, <dc> contains a reference to *bar*. If, in <dc>’s prior existence as an unordered static archive, the definition of *bar* preceded its reference, the *ld* operations to build *x* may have satisfied <dc>’s reference with the definition from <sc>. However, in its dynamic form, <dc>’s own definition will be used. This is a consequence of the fact that at execution time, all searches for a symbol definition start with the main program and then all .so’s in load order. This behavior preserves the ability to *interpose* on library entry points.

4.4. Debuggers

The debuggers used in the SunOS environment, *adb* and *dbx*, have been modified to deal with the dynamic linking environment provided by the new *ld*. In particular, they understand that symbol definitions may appear after a program starts executing. Such dynamically added symbols are found by noting the creation of the *link_map* structure list in the initial program’s *__DYNAMIC* structure, and adding the symbols for the .so’s that have been added to the debugger’s database of symbols.

Despite our goal for transparency in the tools application programmers use, debugger users must also have some awareness of the use of dynamic linking. For example, if they reference the symbol *printf* in a program that uses a shared C library but has not yet started executing, the debugger will fail to find it. If, however, such a reference has been made after the same program has executed far enough to call the program’s *main()*, then *printf* will appear.

5. Policies: Applying the Mechanisms

The previous sections have provided descriptions of our approach to providing a shared library capability through the application of basic mechanisms. We have also described the basic mechanisms involved. In this section, we describe the policies by which we use the mechanisms to build a system that provides and uses shared libraries. In general, the considerations applied in setting these policies were (in decreasing order of priority):

- maximize sharing (resource utilization performance);
- maximize flexibility (enriched environment);
- “Principle of Least Astonishment” (user compatibility).

This is to say that a conflict between something that was completely compatible and something that improved sharing or flexibility, generally favored the latter. However, in many cases, it has been possible to accomplish all three considerations.

5.1. System Construction

To meet the goals for resource reduction in the system, the system itself should be built to use shared libraries, and thus, dynamic link editing. This creates the potential for three sorts of problems:

- deferred errors (the so-called “pintf” problem) that are manifested after the system is installed;
- the potential for chaos if an important shared library is deleted; and
- the potential for security problems with “setuid” programs.

To deal with the problem of deferred errors, a set of programs that are supposed to be self-consistent should be built using the assertion-checking facilities previously described.

To deal with the chaos that would result if (for example) a shared C library were deleted from the system, a number of commands and utilities will not be built with shared libraries. These include but would probably not be limited to: *init*(8), *getty*(8), the shells, *mv*(1), *ln*(1), *ls*(1), *tar*(1) and *restore*(8) – essentially programs that would be necessary to restore the missing library from some other source.

Finally, programs that are built as “setuid” (or “setgid” for that matter) are not built to use shared libraries. Such programs could be easily subverted by incorporating a “trojan horse” into a library on which they depend.

5.2. Dynamic Binding

To maximize the benefits of shared libraries, we have decided to make their use the default by having the default binding mode for *ld* be **-Bdynamic**. This creates the potential for users of the programs *ld* builds to be “surprised” by the special considerations of dynamically linked executables the next time they rebuild their programs. In this case, our preference for maximizing sharing took precedence over the potential for user surprise, a choice we made because we believe:

- most users want the benefits; and
- the mechanisms are sufficiently transparent that the “potential” for surprise is not considered to be the same as “likelihood”.

The greatest impact is expected to be on those users who create programs for shipment to other systems. Such users probably want to be isolated from the various problems that a dynamically linked program can have, and should force their programs to be linked statically. While this may impact existing build procedures, such developers usually take special steps when building production programs (such as removing debugging features and employing extra optimization). The addition of another consideration appeared to be a small cost relative to the benefits obtained by the community through maximizing sharing.

5.3. Use of assertions

To help deal with the potential complexities created by dynamic linking, *ld* has been provided with the ability to validate some assertions about an executable it builds. The assertion checking is invoked with the *ld* flag **-assert**, followed by a keyword argument from one of:

definitions	if the resulting program were run now, there would be no run-time undefined symbol diagnostics;
nosymbolic	there are no symbolic relocation items remaining to be resolved; and
pure-text	the resulting executable requires no further relocations to its text.

Together, these assertions are intended to support the development of production programs by allowing the verification of important properties: for instance that a program will not produce run-time link edit diagnostics, or that a piece of code intended to be a “shared library” is in fact sharable.

5.4. PIC Generation

As has been pointed out, PIC in dynamically linked objects improves their ability to be shared and is thus a more efficient use of system resources. However, PIC executes slower than non-PIC, the degree of degradation being dependent on the number of dynamic indirect references the code incurs. Although refinements to the generated code may ultimately make the performance impact of PIC negligible, we have

chosen to make the use of PIC an option. Our expectation is that only code intended to be part of a shared library will be compiled as PIC.

We also expect that few users will enable the generation of PIC in their application programs, simply because it takes extra effort to do so. However, this raises the issue of the binding of non-PIC code to the PIC shared libraries it uses. The binding that must occur involves all references to:

- **commons:** allocated after the program is completely assembled;
- **initialized data:** imported from the shared libraries; and
- **entry points:** supplied by the shared libraries.

The implication of these binding operations is simply that the link editing that implements the binding will render the edited code unsharable.

To improve the degree of sharing for such programs, *ld* can be made to force the allocation of commons and to create aliases for library entry points. These allocations and aliases are created as part of the non-PIC executable, and result in “pure-text” non-PIC programs even if they have dynamically linked components. These options (**-dc** to force the definition of common storage, and **-dp** to force the definition of procedure aliases), are included by the C compiler driver automatically in the *ld* command line it generates.

6. Examples: *.so* Construction

6.1. Shared C Library

The construction of the shared C library involved:

- compilation of all of its C source modules using the **-pic** option;
- modifications of some assembly-language source files so that the assembly source was also position-independent; and
- *ld*'ing the resulting collection of *.o* files to create `libc.so.1.0`.

The modification of the assembly-language source files was, of course, not a requirement for the library to function – simply to make it more sharable. In this area, we fell short of our goal for having shared libraries be transparent to library source code, though happily the amount of assembly source in the system is relatively small.

The *ld* operation, although in reality involving more complex operations resulting from the way we build the various versions of the C library, is conceptually just:

```
% ld -o libc.so.1.0 -assert pure-text *.o
```

assuming the current directory for the command contained only the *.o* files comprising the C library. Since no entry point is supplied, and lacking any other clue to an absolute address, *ld* simply stops processing after it has combined all the object files, built the *GOT* and *PLT*'s, and performed any intra-library PC-relative relocations. The assertion request will cause *ld* to issue a diagnostic if the library requires further relocation to the code contained within it, a sign that a non-PIC object has found its way into the library.

6.2. *ld.so*

The execution-time link-editor, *ld.so* is built with an *ld* command that has the form:

```
% ld -o ld.so -Bsymbolic -assert nosymbolic ... list of modules ...
```

and is conceptually just like other *.so* files. However, it also involves the use of a special binding control option **-Bsymbolic** and the assertion **nosymbolic**.

Normally when *ld* builds a program lacking an entry point or other absolute addressing information, it is unable to perform its symbolic reduction operations simply because it can not assume that symbols from other executable files will not be added later. However, *ld.so* must be self-contained, or else it would require itself to operate and would otherwise pollute the symbol space of the programs it link edits.

The **-Bsymbolic** flag forces *ld* to perform symbolic reduction operations using the information it has now, leaving only relative reductions to be performed – something *ld.so* resolves as part of its

bootstrapping operations. The result should be a completely self-contained program, in which all symbolic references are satisfied by its own internal definitions. The `nosymbolic` assertion tests whether or not this is in fact the case.

7. Examples: Application Construction

To illustrate the use of the mechanisms by applications users, we will consider several simple examples of application program construction.

7.1. “Hello World”

The classic simple C program is the one that simply prints “Hello world” on its standard output using `printf` and then exits. In an environment where the standard library path includes a `.so` form of the C library, the command

```
% cc -o hello hello.c
```

generates the `ld` command

```
% ld -e start -dc -dp -o hello /lib/crt0.o hello.o -lc
```

This `ld` command will cause the creation of the executable file `hello`. Since the default behavior of `ld` is to prefer the use of shared libraries, `hello` will be built as an “incomplete” executable requiring the inclusion of the library file `libc.so[.v]` (where `[.v]` represents the required version string) at execution time.

When the program is executed, `crt0` will discover the `__DYNAMIC` structure `ld` left behind and map in the execution-time linker, `ld.so`. `ld.so` will map in the appropriate version of `libc.so`, allocate any uninitialized commons required by the program, and cause unresolved procedure references in both `hello` and `libc.so` to call `ld.so`. The user’s call to `printf` invokes such a call, causing `ld.so` to search first the symbol table of `hello` and then `libc.so` for a definition of `printf`. The definition is found in `libc.so` and the `PLT` entry for the original call is updated to cause future references to go directly to `printf`. `printf` internally makes other calls to various parts of the C library, each of these intercepted and relocated by `ld.so`.

Although it might be argued that the relocations of intra-C-library calls could have been optimized by prebinding them. However, this would break interposition, as demonstrated by the next example.

7.2. Interposition

Consider the building of the program `hello` again, this time involving a special library, `libinterpose`. This library, like `libc`, is available in a `.so` form. The command used to build `hello` is:

```
% cc -o hello hello.c -linterpose
```

transparently invoking an `ld` command referencing `libinterpose` before `libc`. `libinterpose` defines entry points for various system calls, such as `read` and `write`, that in addition to invoking the required system call also take various statistics on the use of the system calls they surround.

As before, `ld.so` is invoked and maps in the two libraries, first `libinterpose` and then `libc`. The program calls `printf` requiring a relocation to the entry point in `libc`. Eventually, the code that implements `printf` and its descendents issues a call to `write`.

As previously noted, `libinterpose` defines an entry point for `write`. However, so does `libc`, as the standard interface for the `write` system call. `ld.so` resolves the ambiguity by using the ordering it established when mapping in `.so`’s, which places `libinterpose` first. Thus, `libinterpose` is effectively interposed for all uses of `write` in this program. If `hello` itself had defined a `write` entry point, it would have taken precedence over both `libinterpose` and `libc`.

7.3. Mixing Static and Dynamic Binding

Consider a program linked with two shared libraries, `liba` and (automatically) `libc`. A third library, `libb`, however it is only available in an archive, or `.a`, form. These are combined with the program `foo.c` with the command

```
% cc -o foo foo.c -la -lb
```

`foo` references a procedure `bar` defined in both `liba` and `libb`. `ld` handles this problem by recognizing that `liba` contains a definition for `bar`, and ignoring the one provided in `libb`. Thus, even though the material from `liba` is not incorporated into the program until execution time, `libb` is prevented from contributing a definition.

However, suppose `foo` did not reference `bar`, but `liba` did and further, had no definition for it? In this case, `ld` would incorporate the definition from `libb`, and again the intent of the ordering on the command line is followed despite the difference in binding times.

8. Conclusions and Future Work

We have described the design of a shared libraries facility satisfying most of our goals, including:

- no kernel support specific to shared libraries or dynamic linking;
- transparency to application source code and build procedures;
- transparency to library source in higher-level languages; and
- no administrative procedures required to create or use shared libraries.

Some goals for transparency were only partially achieved, the most significant being the potential confusion to those using the system's debugging tools. The need to change some library assembly source is considered an acceptable minor shortcoming.

Although we have only limited experience with the implementation, early performance measurements indicate that we should meet our performance goals for "average" programs. These early measurements reveal:

- **PIC degradation:** the use of PIC in libraries does degrade execution time, although in many programs the degradation is negligible. The degradation is most noticeable in those programs that execute primarily in the libraries, and in some cases the degradation fails to meet our limits of 10%. However, we believe there are opportunities for improving the generation of PIC.
- **Start-up costs:** programs previously dominated by start-up costs and that use only a few libraries fall within our 50% goals. We have identified several areas for optimizing the start-up process, including caching the results of library searches. The start-up overhead for programs that use many libraries is unacceptably large, and is an area we are investigating.
- **Space reduction:** measurements over most of the system's standard utilities suggest that the average per-program savings from the use of shared libraries will be approximately 24K bytes.

During the time we obtained these early measurements, the new VM system on which the work was performed was also being debugged and shaken-out. The measurements were taken in a worst case environment where only the test programs employed shared libraries. Thus, any benefits or problems created by the dynamics of an environment that is based on shared libraries have not been determined, though it is expected that the sharing of the C library will have a positive impact.

Like most technologies, shared libraries and the mechanisms from which we build them can be abused. The execution-time loading we perform clearly has a cost, and excessive use of it in production programs may produce unacceptable performance. However, extensive use during program development adds a new element of flexibility that developers can use to enhance their development environment. An additional consideration is that a library is now a more powerful construct. Previously, the benefits of libraries were in the packaging they provided commonly used facilities. However, that packaging can now be used to provide performance and functional benefits as well.

Our future plans include:

- **Performance enhancement:** continuing efforts in this area for the foreseeable future. An area of particular interest is work to provide different space/time trade-off points than the two provided by the current implementation.
- **Common Link Editor source:** although they can be conceptually viewed as one, at present the two link editors `ld` and `ld.so` are implemented as separate programs. `ld.so` is particularly simplified, a short-cut taken to speed implementation of a first cut at the facility. We would

like to build both programs from a common source. Ideally, *ld* should just be an executable jacket to the common code in *ld.so*.

- **Programmatic interface:** some programs, particularly based on interpretive languages such as LISP, can dynamically generate dynamic references. We would like to support the handling of such references through a common mechanism, and thus wish to provide a program-accessible interface to the services now provided invisibly.
- **Different exception handling:** the current disposition of execution-time errors is to abort the program in which they occur. We would like to investigate the program development environments that might be created with other exception handling policies.

9. Acknowledgements

David Goldberg worked on the initial design for a shared libraries prototype that was the basis of this effort. Evan Adams and particularly Richard Tuck spent many hours discussing and helping to refine the philosophies of link editing employed in this design. Bill Joy provided advice and helpful criticism as well as several alternative and interesting implementation strategies. Bill Shannon was helpful in reviewing drafts of this paper.

10. References

- [ARNO 86] Arnold, J. Q., "Shared Libraries on UNIX System V", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.
- [DOWN 84] Downing, C. B., F. Farance, "Transparent Implementation of Shared Libraries", *UniForum Conference Proceedings, Washington DC, /usr/group*, January 1984.
- [GING 87] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.
- [MURP 72] Murphy, D. L., "Storage organization and management in TENEX", *Proceedings of the Fall Joint Computer Conference, AFIPS*, 1972.
- [ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.
- [SZNY 86] Sznyter, E. W., P. Clancy, J. Crossland, "A New Virtual-Memory Implementation for UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

A Debugger-based System for Graphical Display and Editing of Data Structures

Peter Potrebic and Phil Goldman

Apple Computer
Mailstop 27AJ
20525 Mariani Avenue
Cupertino CA 95014
(408) 973-3293

ABSTRACT

Certain flaws are inherent in the design and implementation of text-based debuggers. A major shortcoming is that these debuggers cannot provide a simple and efficient interface for data manipulation. A system based on the window and mouse metaphor provides a solution. Such a system allows for more user control over data display, a more intuitive representation of the information needed, and a quicker method for altering data.

Such systems do exist. This paper describes one such implementation known as `gdbxtool`.

1. Introduction

The creation of "correct" programs is one of the central issues of computer science; it is certainly important to those who use the applications of this science. While there has been some theoretical work dealing with the generation and verification of correct programs, this field is still in its infancy. The everyday programmer still has to "debug" his program empirically rather than prove it correct theoretically.

Although debugging is still more of an art than a science, a set of features exist that greatly simplify the practice of debugging. A debugger that implements all of these would be a very powerful tool. The following paragraphs describe these features. Next is a brief survey of existing debuggers, and then the main portion of this paper discusses `gdbxtool`, a debugger with a graphical interface for the display and editing of data structures.

The first important feature is symbolic debugging. A debugger should understand the language in which a program was written so that the level of abstraction a language provides can be maintained when the program is viewed under the debugger. Such a debugger should understand multiple languages; it should not force a programmer to use a particular language.

However, a debugger should also support a lower level of debugging. In this way a programmer can decide on what conceptual level he wishes to debug his program. For

example, during the course of tracking down a bug the compiler might become suspect. In this case it is useful to examine the object code, rather than the high level language.

Another useful feature is the ability to latch onto an existing process and debug it in its current state. Often, it is very difficult to reproduce a bug after restarting the process under the debugger. The ability to debug existing processes also allows for the debugging of processes that must be created when a machine boots up and exist over the lifetime of the machine. It is easier to "fix" a wayward process, by means of the debugger, than it is to reboot the machine in order to restart the process under the debugger.

The ability to debug multiple processes is another valuable function. For instance, a distributed database may be divided among many cooperating sub-processes. To effectively debug this application the programmer must be able to debug any or all of the sub-processes, and also observe the flow of data among them. This ability should extend to arbitrary processes, not just sub-processes of some parent application, and should span machine boundaries.

Flexibility is an essential characteristic of any debugger. The user should be able to customize the debugger's data display and command macros. Just as the programmer has developed a local language in the domain of the source language, so too should he be able to create a local language in the domain of the debugger commands. In the Unix¹ world the shell languages are used to increase the power and flexibility of the environment. A debugger with a similarly powerful language would allow a programmer to build a set of functions to aid a particular style of debugging.

Another feature of a powerful debugging tool is the ability to present data in a fashion that is both meaningful and intuitive to the user. Why should the programmer be forced to format data, when the debugger could easily do so? When debugging code involving a hash table, the developer needs to see the data structure as visualized, in a graphical fashion, and should be able to edit the data with a graphical interface as well. Also, the developer should be able to customize the display of the data as desired. This feature is implemented in **gdbxtool**, the debugger discussed in this paper.

The final feature is speed. A debugger that implements all the features mentioned above but runs too slowly is useless. The debugger environment, language and data display must be efficient so that throughput and response time to user commands are acceptable.

The next step is a brief survey of existing Unix debuggers, concentrating on the above features.

Of the mainstream debuggers, **adb**² is the most primitive, but it does have its strong points. It is fast and gives the user access to a low level of debugging. Unfortunately, **adb** cannot be used on the symbolic level. More recent debuggers such as **dbx**³ allow for symbolic debugging.

It was not until the introduction of **dbxtool**⁴ that any attempts were made to improve the interface and add graphic capabilities to **dbx**. This program is a mouse- and window-based interface between the programmer and the debugger. New features included a few user-defined commands, and the ability to customize the display. The designers realized

¹ Unix is a trademark of AT&T Bell Laboratories.

² 4.1BSD Reference Manual.

³ 4.3BSD Reference Manual.

⁴ Evan Adams and Steven S. Muchnick, "Dbxtool: A Window-based Symbolic Debugger for Sun Workstations", 1985 Summer USENIX Technical Conference Proceedings, 213-227.

that graphical display of variables was desirable, but at the time they felt that such a feature would not be feasible.⁵ An earlier effort added graphical display of data to *dbxtool*⁶, but the feature was too expensive because a separate process was used to handle the graphics.

Two additional symbolic debuggers are *joff*⁷ and *Pi*. Both debuggers have a powerful user interface that includes windows, menus, and mouse input. *Pi* has many desirable features including the ability to latch on to an existing process and the ability to debug multiple processes. The major limitations to *Pi* are that it lacks a powerful command language and graphical display of data. The former was a conscious omission on the part of the *Pi*'s developer who instead concentrated on a powerful user interface.⁸

All of the debuggers mentioned above implement some, but not all, of the given desirable features. Most noticeably lacking is the graphical display and editing of data and a powerful command language. The lack of graphics hardware at the time most of these debuggers were developed partially explains why none of them attempts to display data in a graphical fashion. However, with the low cost of today's graphics workstations and windowing software much more sophisticated solutions are possible. Any such solution should include enough flexibility to allow for the fact that what is intuitive to one user may not be to another. In practical terms this translates into the implementation of user-defined macros and a flexible display format.

2. Desirable Display Qualities

As mentioned above, the field of debugging lacks a theoretical foundation. However, there is one aspect of debugging where common sense can show what properties are required for an effective tool. This is the area of program data display. The following is a partial list of these rules for effective data display:

- **Display all relevant information simultaneously.**

Often, a programmer sifts through a great deal of data and state information to find the cause of the bug. A good debugger must allow the user to display all the relevant information while filtering out useless data, in order to speed the process of tracking down bugs.

- **Retain information on the screen.**

Many debuggers are based on a line-oriented interface where a single region is used for command input, display of data, and other state information (i.e. trace information, program flow, etc.). Usually this region automatically scrolls so that information is lost off the top of the screen. This type of interface is undesirable because the user has no way to keep important information on the screen. A solution is to have separate display regions which do not automatically scrolling. With such an interface commands are entered defining what information is to be displayed. The information remains in the display region, and is periodically updated to reflect current values.

Dbxtool has such a display region, for variables. However, one drawback is that all variables, whether simple scalars or more complex data structures, are displayed as

⁵ Ibid, 214.

⁶ David B. Baskerville, *Graphic Presentation of Data Structures in the DBX Debugger*, Unpublished manuscript, 47.

⁷ Cargill, T.A., *Debugging C Programs with the Blit*, AT&T Bell Laboratories Technical Journal, October 1984, Vol. 63 No. 8, Part 2, 1633-1647.

⁸ Cargill, T.A., "The Feel of Pi", 1986 Winter *USENIX Technical Conference Proceedings*, 62-71.

text in a line-oriented manner. Another drawback is that the user cannot reorganize the display except by inserting and deleting variables to control their vertical ordering.

- **Manipulate data in a simple fashion.**

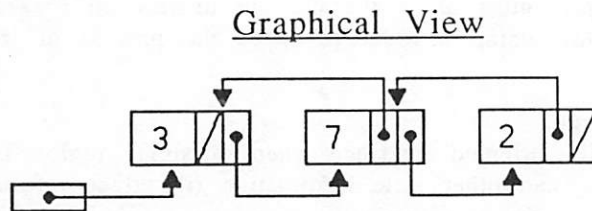
Text-based debuggers often violate this rule. The most common example of this is the manipulation or display of a member of a linked list. The only way to reference an element of a list is to use its location in the linked list, assuming it has no explicit name of its own. Therefore, the length of the name the programmer must type is proportional to how far the element is removed from the head of the list. For example, the command to display the sixth element of a list might be

```
display *head->next->next->next->next->next
```

Where *head* is a pointer to the first element in the list. The fact that the linked list is seen as a sequentially accessed structure by the machine does not imply that the programmer should be forced to access it sequentially when debugging.

- **Display data as the programmer visualizes it.**

This is desirable as it allows the programmer to quickly determine the state of the system he is debugging. This is especially important when displaying a large amount of data. An example of poor display is the way in which many debuggers present pointers, as numerical values (often in hexadecimal or octal) of the address of the object to which they point. The ideal format would clearly show the relationship between the



Textual View

```
head = 0x3a4d8
*head = {
    value = 3;
    prev = nil;
    next = 0x42ad2;
};
*head->next = {
    value = 7;
    prev = 0x3a4d8;
    next = 0x4fd38;
};
*head->next->next = {
    value = 2;
    prev = 0x42ad2;
    next = nil;
};
```

Figure 1.

pointer and the object it references, but this is difficult to accomplish with text. If all objects have names, pointers could be displayed as "pointing to *x*", for object *x*; however, many languages allow for dynamic allocation of objects. These objects do not have explicit names in the source files; they are referenced solely through pointers. Therefore, the debugger would have to create a new name for each object dynamically allocated. This still does not allow the programmer to visualize the relationship between pointer and object. Figure 1 compares the two methods for displaying linked-

lists. The graphical view is superior because it clearly shows the structure of the data.

- **Allow the user precise control over display format.**

For example, the programmer should be able to specify which fields of a data structure are visible. The format control could apply to a specific variable or to all variables of a given type. For instance, in a 100x100 array, display only the middle 20x20 entries. Further, the programmer should be able to position variables in the display area in relation to associated data structures, or by absolute position.

- **Speed.**

This is always an important factor. The data display should not noticeably slow the debugger. The reality is that there is always a tradeoff between speed and complexity.

As can probably be guessed, most of the desirable features listed above exist in `gdbxtool`.

3. Gdbxtool

The purpose of this paper is to report on the experiences of the authors in their design of a debugger interface that displays data graphically. As this study was intended as research rather than product development, the interface was built on top of an existing debugger, `dbxtool`.^{*} Although the resulting tool was less efficient than if designed from the ground up, the authors did not have the resources to implement a complete debugging environment. We felt that `dbxtool` had taken some steps towards a more intuitive approach to the display of data and so decided to use it as a starting point from which to develop a graphical interface to the display and editing of data structures.

The authors designed and implemented a graphical addition to `dbxtool`. `Gdbxtool` runs as a modified `dbxtool`; it is not a separate process. It replaces the text-based display subwindow with a graphics-based subwindow where variables appear in their graphic form.

The display routines are built on top of a device independent graphics package, *SantaFe*.[†] This gives users of `gdbxtool` the ability to display their data structures on various graphics devices and to print out the pictures on different output devices.

Based on the desirable qualities listed above, the authors' goals for `gdbxtool` were:

- Allow for easy and efficient means for the display of all types of program data.
- Create a simple mechanism to modify data.
- Develop an educational tool where users can actually see algorithms at work, for instance, linked lists being manipulated.
- Allow the user precise control over which data is viewed and how it is displayed.

3.1 Data Structures

`Gdbxtool` treats all variables as graphic entities, in the *box and arrow* format. Simple scalars are drawn with the name followed by the value both surrounded by a box.

^{*}The authors would like to thank Steve Muchnick and Sun Microsystems for their cooperation.

[†]*SantaFe* was developed at Princeton University under the supervision of Professor David Dobkin.

Structures (or records) are drawn with the structure name on top and the fields drawn indented underneath. Arrays are drawn the same way as structures, with the array name at the top and array indices as field names. This array format creates a sense of consistency in that the user does not need to learn different formats for arrays and structures. See figure 2 for examples of how different types of variables are drawn.*

Nested data structures and multi-dimensional arrays are drawn recursively. A nested structure is displayed as described above, only it is drawn within its parent structure. Each level of nesting (or each added dimension of an array) is indented down and to the right (see figure 2). This allows the user to easily visualize the layout of variables.

The user can also edit the values of data displayed on the screen. Instead of typing the `dbx set` command followed by the variable name and new value, the user simply clicks on the variable, or the field in an array or structure, and types a new value. This makes modification of data extremely quick.

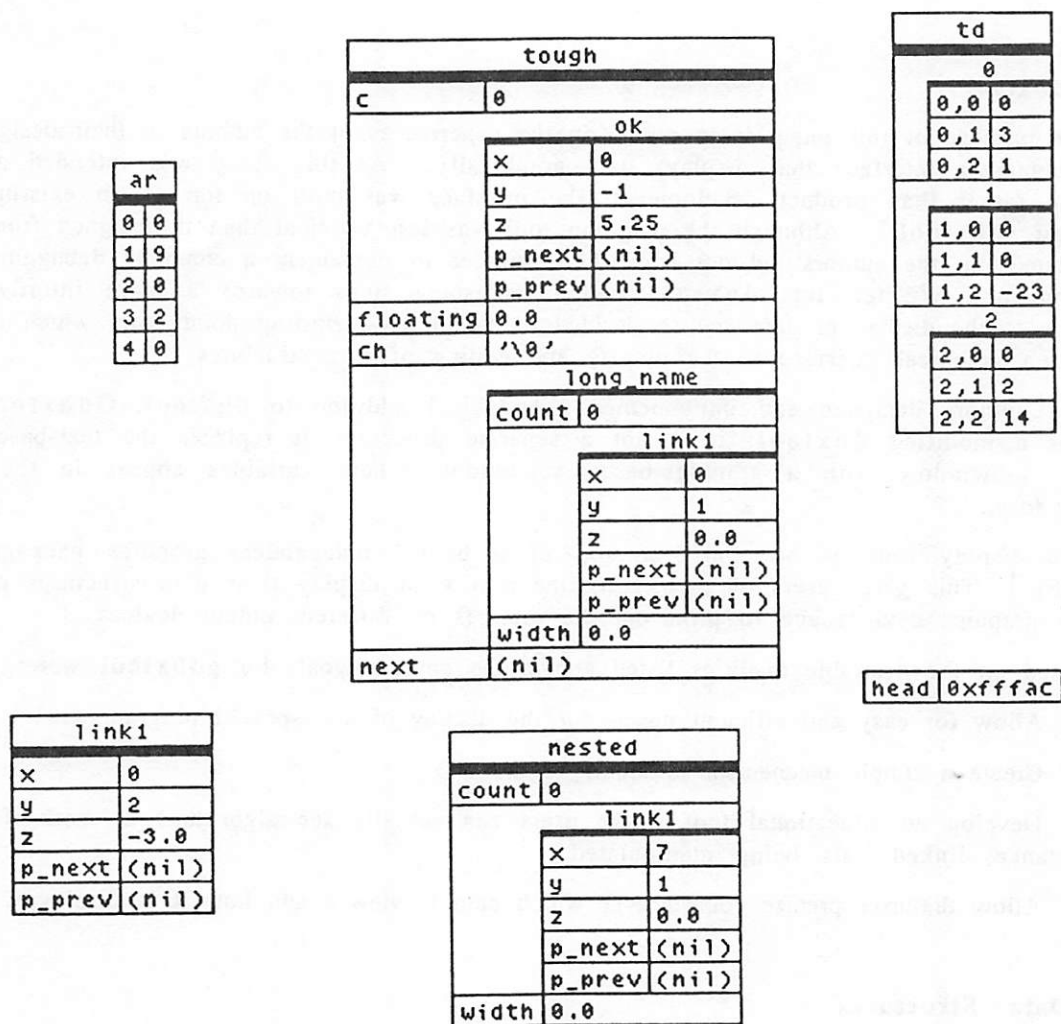


Figure 2. Displayed variables

* All figures from this point on are actual screen dumps created using `gdbxtool`.

Gdbxtool allows the user to use the mouse to open and close variables and fields within structures or arrays and to move variables. When an entire variable is closed only its name is displayed. If the variable is later re-opened it is redrawn in its previous format. Fields can also be opened and closed. Closing a field causes the variable to be redrawn as if the field did not exist, with one exception. The horizontal bar below a structure's name indicates which of its fields are open. The bar is divided into a number of segments equal to the number of fields. Each section is painted black if the corresponding field is open and white if closed. This applies to nested structures as well (see figure 3a).

tough	
c	0
ok	
y	0
z	0.0
p_prev	(nil)
ch	'\0'
long_name	
count	0
link1	
x	0
y	0
p_next	(nil)
p_prev	(nil)
width	0.0
next	(nil)

figure 3a

struct h	
c	int
struct link	
x	int
y	int
z	float
p_next	struct link *
p_prev	struct link *
floating	float
ch	char
struct links	
count	int
struct link	
x	int
y	int
z	float
p_next	struct link *
p_prev	struct link *
width	float
next	struct h *

figure 3b

```
struct h {
    int c;
    struct link ok;
    float ch;
    struct links long_name;
    struct h *next;
};
```

```
struct links {
    int count;
    struct link link1;
    float width;
}
```

```
struct link {
    int x, y;
    float z;
    struct link *p_next, *p_prev;
};
```

```
struct h {
    int left, right;
}
```

figure 3c

Figure 3. A variable, its template, and C definition

Opening and closing a data structure does not affect the state of its fields. For example, if a variable is closed and then re-opened the same fields will be visible as were before the variable was closed.

All variables and all data types have an associated template. This template looks similar to the variable display but is used to display and manipulate information on a more global level (see section 3.4). Figure 3b shows the associated template for the variable in figure 3a.

3.2 Pointers

Another area of debugging that lends itself to the use of graphics is pointer display. **Gdbxtool** draws all existing pointers between variables and/or fields as arrows with the head of the arrow at the location of the object pointed to and the tail at the location of the pointer itself (see figure 4). The shaft of the arrow is a set of connecting line segments – one line segment would be simpler but would cause arrows to be drawn on top of variables, an undesirable effect. If the value of a pointer is not the address of an object currently displayed, the numerical equivalent of the pointer value is displayed rather than an arrow.

It is somewhat difficult to guarantee that the pointer arrows never intersect variable boundaries. Not only does **gdbxtool** have to redraw pointers to and from data structures that are displayed, undisplayed, moved, and resized (due to the closing or opening of the structure or its fields), but it must also redraw all pointers from other variables that were indirectly affected. For example, say structure A has a field that points to structure C and the arrow is drawn around structure B. If structure B is moved then the arrow between A and C may have to be redrawn. This is a non-trivial graphics algorithm as there is no efficient way to encode the fact that B lies between A and C. Worse yet, it is extremely difficult to determine what the optimal arrow shaft would look like; there might be structures D,E,F,...,Z that also lie between A and C.

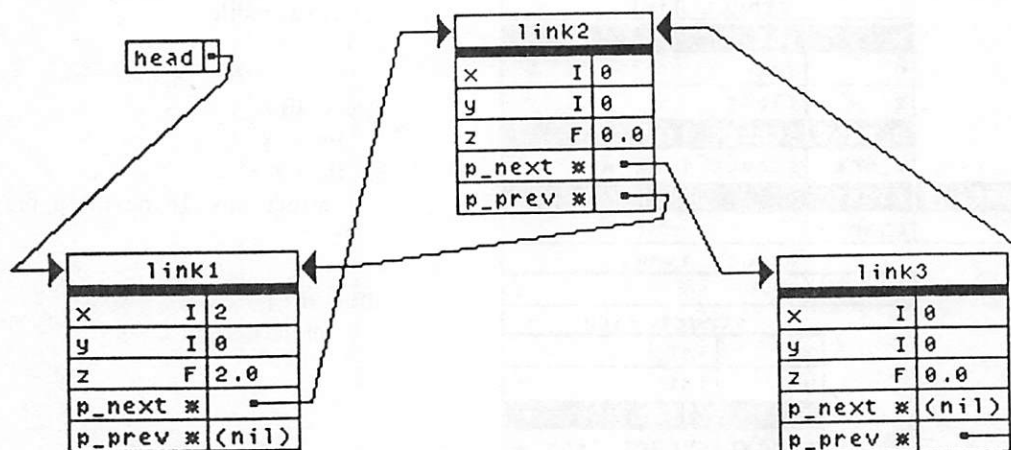


Figure 4. Pointers

A heuristic algorithm for drawing arrow shafts has been devised, based on grouping overlapping data structures together and drawing the shafts around these groups. Each data structure in the display region belongs to exactly one *hull rectangle*. Each of these rectangles contains one or more data structures; the hull is defined as the smallest rectangle that will completely surround all of its member data structures. When a new

data structure is displayed a new hull is created with that variable as its only member. If that hull intersects any other hulls then these are combined. Similar rules apply when other operations (i.e. moving, resizing) are done on the members of a hull. The basic concept behind these rules is to guarantee that no two hull rectangles ever overlap. Then, the algorithm for drawing arrow shaft is reduced to finding the shortest path around rectangular obstacles. This path can be constructed out of connecting line segments as the obstacles themselves are never curved.

3.3 Multiple Views

Gdbxtool also takes advantage of the fact that **dbx** allows multiple instances of the same variable to be displayed. Previously, this was a trivial feature, but with the added support **gdbxtool** provides it becomes much more useful. Figure 5 show the same variable being displayed twice. The first instance only shows the last two fields while the second shows the first three.

link1	
p_next	0x10d5c
p_prev	(nil)

link1	
x	2
y	0
z	2.0

Figure 5. Multiple views

Consider the same variable displayed in two different contexts. One instance is displayed as part of one linked list while the other is displayed as an element of a second linked list. The ability to separate the two lists visually, even though they share the same variable, greatly clarifies the relationships displayed on the screen. Consider a program that manipulates vertices and edges. A single vertex may be a component of several edges. There may be times when the developer wants to see the various data structures associated with two particular edges (its component vertices) but does not want the vertex data structures to appear shared.

3.4 Templates

Yet another feature of **gdbxtool** is the ability to define and alter templates. Each variable has a corresponding template, and each data type has a template that applies to all variables of that type. This gives the user the ability to customize the display of data structures.

The template defines how a variable or group of variables is to be displayed; it determines which fields are closed, if data type letters should prefix field names (see section 3.5), and other related characteristics. By manipulating the appropriate template the programmer can quickly close all fields that are not of use, freeing precious space on the screen for more useful information. Once a template for a given type is defined all current and future variables of that type are drawn according to the definitions of the template. Of course, the user can adjust an individual variable by modifying its private template, overriding the template for the particular data type.

Basically, the template is a graphical representation of the variable definition that normally appears in the source header files (compare figures 3c and 3b). So, if the user wants to determine the type of a data structure he can display the variable's template rather than searching through the source files. Further, the graphical representation is more intuitive and informative than the source.

3.5 Type Prefixes

Gdbxtool can display a short type abbreviation for all fields and variables. The abbreviation appears immediately preceding the variable or field name in the same box, as shown in figure 4. Integer fields are preceded by an "I", floats by an "F", pointers by a "*", and so on. This function can be turned on or off by manipulating templates, again either for individual variables or for a given data type. This feature helps identify variable and field types so the user need not search through the source code or bring up a template just to determine type.

3.6 User Interface

The functionality described above is a start, but the user must have easy access to these capabilities if they are to be useful. **Dbxtool** has a mouse-driven interface and **gdbxtool** extends that interface to include several new features. **Gdbxtool** has its own menu, panel, and event mechanisms, which were implemented using *SantaFe*. The following is a brief overview of this user interface.

Gdbxtool uses pop-up menus very similar to the SunWindows menus. There are a total of three menus; the one displayed depends on the location of the cursor when the mouse button is pressed. The main menu appears over the background, the variable menu appears over the box around a variable's name, and the field menu appears over a structure's field (see figure 6). These menus only appear in the variable display subwindow.

There are three different panels that appear depending on the state of the system. The main panel can be seen at the bottom of figure 6. The panels are static menus that further reduce the dependence on keyboard input by increasing use of the mouse. Panels are used rather than menus for operations that either affect all the displayed data or require further input, via dialog boxes, to determine the exact course of action.

Although using the mouse can be more efficient than typing, users often get frustrated when multiple clicks in different regions of the screen are required to perform simple tasks. **Gdbxtool** has several accelerators that allow experienced users to quickly carry out common operations. One click will open or close variables or close fields (rather than using the pop-up menus). One click allows the the user to change the value of any variable, as opposed to using the **dbx** command *set*. Finally, moving a variable takes one click and drag, rather than using the pop-up menu and then clicking and dragging. These shortcuts vastly improve performance.

One of the goals of **dbxtool** was to reduce the dependency on the keyboard as an input device. This was accomplished by adding user-definable buttons, the ability to point and click on the actual source, and the display subwindow. **Gdbxtool** has this same goal to reduce keyboard dependency. The goal has been achieved; a programmer using **gdbxtool** will do less typing than if he were to use **dbxtool**, and much less than **dbx**. As a result debugging is more efficient and less time-consuming.

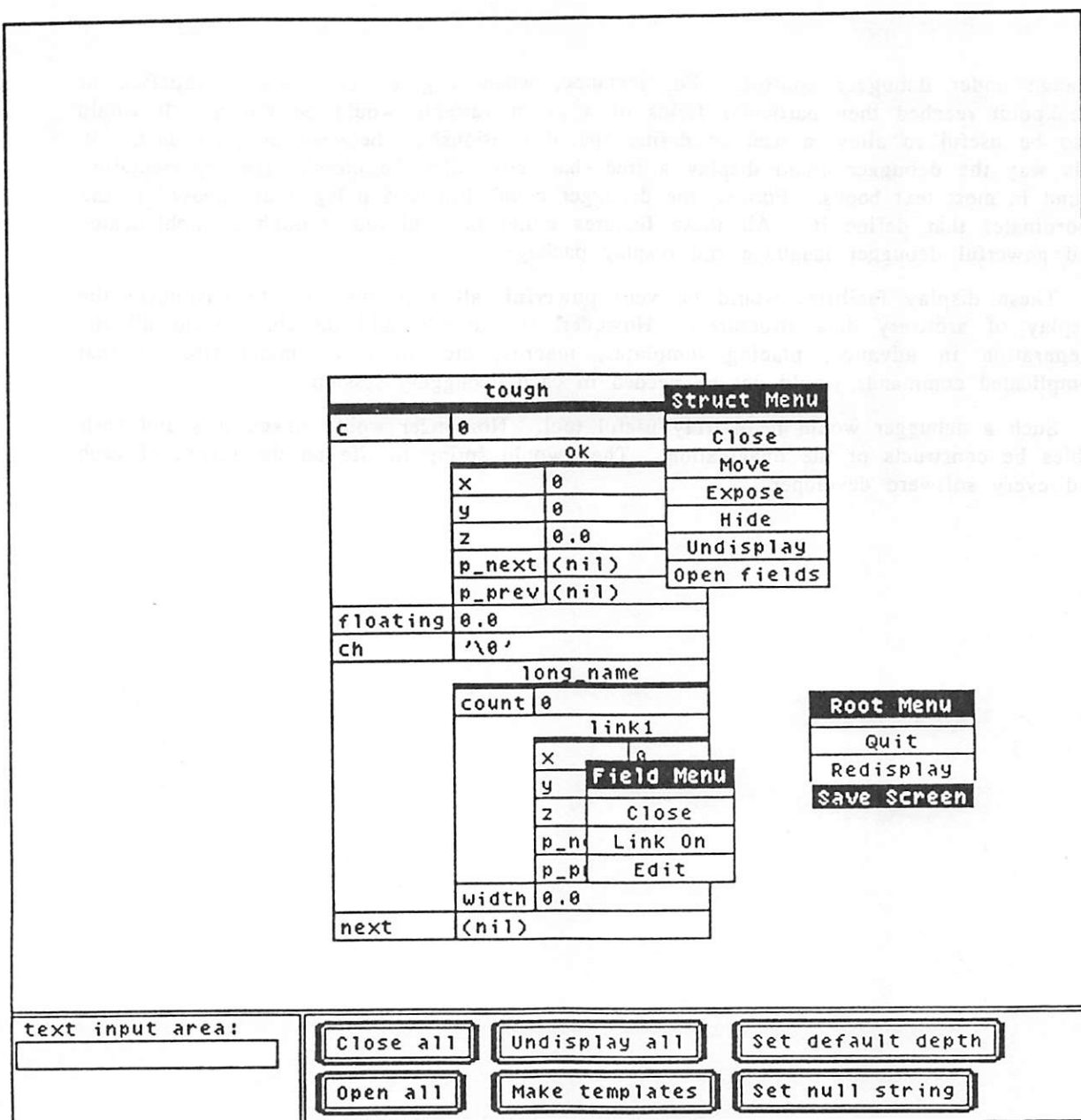


Figure 6. Menus and a Panel

4. Future Work

We have demonstrated here how flexible display of information greatly enhances the power of a debugger. As previously mentioned, graphic display of data should be considered when the debugger is being designed, not added as an afterthought. Our work here, coupled with faster machines, justifies the design and implementation of such a display-based debugger.

In addition to the implementation of concepts introduced in *gdbxtool*, a display-based debugger would benefit from certain other extensions. Templates are a first step towards a macro language to manipulate the display of data, giving the user a static capability to control the display. However, it would be useful to allow a template to

change under debugger control. For instance, when a given condition is satisfied or breakpoint reached then particular fields of a given variable would be shown. It would also be useful to allow a user to define spatial relationships between program data. In this way the debugger could display a tree that looks like the classic tree representation found in most text books. Further, the debugger could display a polygon as opposed to the coordinates that define it. All these features could be achieved through a sophisticated and powerful debugger language and display package.

These display facilities would be very powerful, allowing the user to customize the display of arbitrary data structures. However, the user should be able to do all the preparation in advance, placing templates, macros, etc. in a command file so that complicated commands would not be needed in each debugging session.

Such a debugger would be a truly useful tool. No longer would linked lists and hash tables be constructs of the imagination. They would spring to life on the screen of each and every software developer.

A New IPC System for Bitmap Graphics Applications: Review, Model, and Benchmarks

C. Douglas Blewett
Myron (Mike) Wish
Jonathan I. Helfman

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

One of our research goals is to develop a suite of modular, interactive graphics tools, along with an approach for integrating subsets of them into larger systems. The hope is to enhance the computing environment and interface for general, as well as for sophisticated UNIX users. This work has been motivated by our interest in multiprocess systems that act as one application, as well as related work on systems that share data.

The backbone of our current effort is a prototype IPC system for processes running in a graphics engine, the AT&T 5620 Dot Mapped Display terminal. This environment seems well suited for creating multiprocess interfaces and quick prototypes since it avoids the high overhead associated with more expensive operating systems. Such an IPC system also encourages modularity and data encapsulation, partitioning a problem into small, manageable, reusable parts.

The paper begins with a review of some other IPC approaches, along with a set of design considerations for IPC to work well in a bitmap graphics environment. The details of our model and implementation are then discussed and illustrated, followed by benchmark timing comparisons with other UNIX-based approaches.

1. Introduction

1.1 Perspective

About a decade ago Hoare [10] proposed that interprocess communication (IPC) be considered a primitive for programming. He outlined a scientifically rigorous language approach, called *communicating sequential processes* or *CSP*, replete with elegant syntax and semantics. This included synchronous and nonbuffered message passing between processes, the fundamental units of concurrent program execution. The underlying hardware envisioned and modeled was a loosely coupled distributed network of processors.

Current interest in object-oriented programming, particularly Smalltalk [9], has reinforced Hoare's view that message passing should be an organizing principle for language and system design. In Smalltalk, arguments to a method (more like a procedure than a process) are sent as messages. Even control structures are accomplished by sending messages to

objects. Message passing, which is absolutely central to the language, is extremely flexible, including the ability to send code fragments to be executed in the context of the receiving process.

Message passing, and more generally interprocess communication, is now an integral element of forward-looking work in such subspecialties as programming languages [8,15], computing environments, and operating systems. Furthermore, within the last year two new bitmap graphics systems [1,2] have been produced that rely on message passing to perform all operations. As each substantive area has different requirements for communications, the assorted IPC systems produced have presented surprisingly varied approaches to the problem.

But what does this burst of interest in interprocess communication contribute to modern computing environments? One of the better answers is that IPC encourages modularity and data encapsulation, partitioning a problem into small, manageable, reusable parts; and systems constructed from communicating processes are generally built from parts at a higher level of conceptualization than are software monoliths.

Although UNIX[®] is famous for its ability to solve big problems by combining small programs with pipes, there are several limitations. Pipes require a common parent somewhere up the process hierarchy, and are not really suited for connecting interactive programs. Moreover, modern, multiprocess applications typically have needs for passing variable-sized, but structured, data rather than simply the character stream provided by pipes. Various versions of UNIX, as well as the newer operating systems [7,11], have IPC facilities for addressing such requirements.

Our own concern is to create an environment and approach for building bitmap graphics applications, based on cooperating modular processes. Toward this end we have designed and implemented a prototype IPC system that runs in a graphics engine, the AT&T DMD 5620. Before describing the details of our IPC system, however, we would like to review some other approaches and to propose some design considerations relevant for a bitmap graphics environment.

1.2 Other IPC approaches

Berkeley software distribution (4.2 and 4.3) includes *sockets*, an inter-machine scheme intended to be the end-all for interprocess communication. There are five types, *stream*, *datagram*, *raw*, *sequenced packet* and *reliably delivered message* sockets. Only *socket streams* have been made widely available, and the last two types have not yet been implemented. There are also remote procedure call packages [4] built on top of Berkeley sockets, but they will not be discussed here.

Sockets are based on a complex model, incorporating many different naming conventions and protocol handlers, but for most purposes names correspond to files on one of the locally mounted file systems. Socket streams, the protocol in most cases, are bidirectional and reliable, behave essentially like pipes for reading and writing, and allow for sequenced and unduplicated flow of data without record boundaries. Variable-sized messages are handled by agreement between the cooperating processes. *Datagram* (structured message) sockets, which preserve record boundaries in data, are not guaranteed to be sequenced, reliable, or unduplicated. *Raw* sockets, which are normally datagram oriented, are intended for developing new communication protocols.

The interprocess communication in UNIX System V (releases 2 and 3) is a simple, single-processor scheme based on structured, variable-length messages. (We do not discuss here other System V mechanisms such as pipes, named pipes, and semaphores). Both rugged and efficient, it requires only a modest number of lines of C code compared to Berkeley sockets. All messages are required to have a type value, which may be used by the receiving process as a selector. *Fifo* order is otherwise preserved. Sending processes cannot specify the receiving process nor vice versa.

The Ninth (and Eighth) Edition AT&T research versions of UNIX come with stream-based IPC [14], which provides convenient ways for programs to establish communication with unrelated processes on the same or different machines. A *stream*, which works essentially like a two way filter, is a full duplex, kernel supported mechanism for connecting user processes to each other as well as to devices or pseudo-devices. Communication between linearly connected processing modules is primarily accomplished by passing messages to neighbors in the two-way pipeline. On top of streams, Ninth Edition IPC looks something like a file that behaves like a pipe (or socket stream) for reading and writing — write and I/O control requests turn into messages that are sent to the stream, while read requests pass data from the stream to the user. They differ from sockets in that message (write) boundaries are preserved.

Mach [11] and *V* [7] are two new operating systems that give particular attention to interprocess communication. *Mach*, which is currently under development at CMU, is an IPC-based operating system with distributed, variable-length messages. The kernel interface is defined in terms of RPC's. *Mach* can run binaries from Berkeley UNIX systems.

The *V* kernel, which is reminiscent of Hoare's communicating sequential processes, is a system currently being constructed at Stanford to be used on diskless workstations. The basic model is that of synchronous message passing (as in procedure calls) with fixed length messages. Minor additions (i.e. *MoveFrom*, *MoveTo*, *ReplyWithSegment* and *ReceiveWithSegment*) have been included to improve remote disk performance.

There are also application development systems relying on some form of IPC as an organizing principle. The latest of these for bitmap graphics applications are the *X* [2] windowing system from the MIT Athena project and *NeWS* [1] from Sun Microsystems. These systems are based on a graphic server model, that is, applications send messages to the window server to perform graphics related work. The IPC between user supplied processes is that available on the local host machine (usually sockets); this may complicate matters by adding another protocol on top of an existing system.

1.3 IPC features for a graphics environment

This overview of some major efforts in interprocess communication reveals a diversity of goals and needs, as well as underlying differences of opinion about features and implementation details. Our own focus on bitmap graphics applications comes closest to that of *X* and *NeWS*, but has been influenced by the other work as well. In fact, our IPC scheme has its origins in a Lisp-based system, called *Aegis* [6], that has much in common with the approach later taken in *NeWS*.

Having reviewed a sample of the relevant literature, we now discuss some considerations important to us for IPC system design. This is based on our general perspective and experience, and strongly oriented toward working well for bitmap graphics applications,

- Blocking style:** Bitmap graphics applications tend to require non-preemptive, non-blocking (i.e., asynchronous) operation. For example, a paint program should not be preempted while tracking the mouse. Furthermore, if a graphics application has to communicate with another process, it should not be blocked while waiting for the other process to receive the message. Synchronous communication, however, is often used by implementers to avoid making copies of the message.
- Buffering:** In an asynchronous message passing system, applications commonly use free buffer space (e.g. automatic variables) for sending messages. This means that the buffer space in the program may be corrupted by the time the message is received. The system in these cases must make a copy of the message, with copying transparent to the application program. Application programs should not have to include specific buffer management code.
- Message size:** File names, variable names, strings, and other variable-length objects are common items to be sent between cooperating processes. Although some systems, like the V kernel, use fixed sized messages, variable-sized messages are more attuned to application developers' needs. Of course, preferences may differ if flexible sizing of messages leads to a slower implementation.
- Data structures:** The data structures that support message sending should be flexible enough to allow arbitrary communication. This flexibility should not defeat structured communication, a major reason for using IPC. Likewise, processes should not have to perform expensive parsing operations for each message.
- IPC ownership:** IPC should be a capability associated with processes rather than an extension of a file system. In this sense interprocess communication only has meaning as messaging between two active agents.
- Selection:** There should be some way that the receiving process can prioritize messages. Although *fifo* should be the default, selection by other criteria, such as type, tag, or process id, should also be possible.
- Model complexity:** Simple things should be easy to do, with more complex things only slightly harder. A good IPC system should be capable of setting up a connection and transmitting messages with a minimum of code and hassles. It must also include some type of name service as well as other facilities for inquiring about the status of various system elements. Of course, all of this must also be efficient enough to be useful for interactive applications.

2. A 5620 Based Prototype

2.1 Functional overview

Our prototype system has been implemented on the AT&T DMD 5620 previously known as the *blit* [13]. Although the operating system of the host is the Ninth edition research

version of UNIX, there is another small real-time operating system, *muxterm*, that runs the IPC code in the terminal itself.

This environment seems well suited for creating multiprocess interfaces and quick prototypes since it avoids the high overhead associated with more expensive operating systems. The 5620's use of open addressing, one flat unprotected address space, allows new code to be added to the kernel while it is running. It has no file system, and does not run *fsck* when rebooted.

This IPC system was designed to work in harmony with the Pike graphics process model; that is, it has asynchronous message passing, and matches the non-blocking, non-preemptive operating system style of the 5620. (Of course, a synchronous message passing discipline could also be constructed using our system.) For example, a process can accurately track mouse gestures while sending messages and occasionally polling for messages.

A process can send a message to a specific or generic server process. Specific processes are identified by individual process id's, while generic processes are referred to by agreed upon symbolic names. An IPC name service is provided to allow applications to determine which processes are active. These features allow server and user processes to be added dynamically. For example, a user process may wish to spawn a server process if there is not one already active.

In order to allow for long, multi-command style messages, the message data field is implemented as a null-terminated linked list of name-value pairs. This proves to be a useful feature for sending synchronization information when a new server is started. The complete message is copied when sent, which allows the sending process to continue without waiting for the message to be received.

Message delivery is guaranteed to be in *fifo* order. The receiving process may also select messages by tag, generic id, and specific process id (a unique identifier). There is also a facility for monitoring when a message has been handled by the receiving process.

An application may request to be scheduled when messages arrive. When the IPC system is used in this event-oriented style, messages are handled as any other 5620 resource (e.g. the keyboard or the receive queue). Pike's model for handling resources is similar in function to the *select* kernel call.

The intention is to make the IPC code unobtrusive so that the modular nature of existing well constructed programs will not be lost. This should support writing in the UNIX tradition, namely small, modular programs that can be simply combined. Adding message handling to an existing well-structured program can be accomplished by adding three or four lines of code and a switch to sort through the messages of interest. Each program usually contains an initialization line, send and receive lines, and optional calls to wait for messages to arrive. In processes that already contain resource monitoring calls, *ipc_wait* is a modification to the existing call.

2.2 IPC data structures

Three data structures are used within our IPC system: *msg_queue*, *message*, and *msdata*. The *msg_queue* structure contains pointers to *message* structures (functioning as heads of queues), which in turn include pointers to the structure for the message data, *msdata*. There is one *msg_queue* structure per process.

```

/*
 *   message queue
 */
typedef struct
{
    struct Proc    *proc;
    int            id;
    int            signature;
    int            waiting;
    message        *sent;
    message        *received;
    message        *freeable;
} msg_queue;

```

Figure 1. The *msg_queue* (message queue) structure

Each *msg_queue* structure is initialized when the associated process logs into the system. The *proc* field points to the process' proc-table entry, while the *id* field is the specific or generic server id by which the process is known. The *signature* and *waiting* fields are used internally to preserve the sanity, i.e., internal data structures and consistency, of the system. The last three entries in the structure are queues for messages in various states, i.e., *sent*, *received*, and *freeable*.

The first message queue, *sent*, contains the list of messages that have been sent, but not yet received. The *received* queue contains those messages that have been received, but not yet released by the receiving process. Finally, the *freeable* queue contains those messages that have been read and released, and are now freeable. These queues may be used to closely monitor the progress of a message through the system. Access to these queues is handled via requests for name service.

```

/*
 *   message
 */
typedef struct
{
    int            type;
    int            tag;
    struct Proc    *proc;
    int            id;
    struct msdata  *data;
    int            signature;
    struct message *next;
} message;

```

Figure 2. The *message* structure

The *message* structure is used to store messages in the message queues mentioned above, and is returned by the IPC system to the application for all transactions. Error codes are

returned in the *type* field. The next three fields, *tag*, *proc*, and *id* are used to prioritize message delivery. Any combination of these may be specified by the receiving process, and the first message matching the specification is returned.

The *data* field is the head of the linked list of *msdata* structures containing the message data itself. Again, the *signature* field maintains the sanity of the message structures. The *next* field links the list of messages in the message queues. These fields are maintained by the IPC system, and are protected from accidental tampering by the application.

```

/*
 *   message data
 */
typedef struct
{
    int          action;
    char         *name;
    int          length;
    struct msdata *next;
} msdata;

```

Figure 3. The *msdata* (message data) structure

The *msdata* structure is the element of a null-terminated linked list, pointed to from within the *message* structure. In a directory browser application, for example, *action* corresponds to file operations such as open and close, while *name* refers to the filename. (In other applications, "name/value" might be a better choice than "action/name.") In general, the *name* part of the structure is a pointer that can be cast to an arbitrary sized chunk of memory.

The *length* part of the structure is used to determine the size (i.e., for copying) of the element pointed to by *name*. If length is zero, the name is copied as a pointer-sized quantity and no memory is allocated to hold the value.

2.3 5620 library interface to IPC services

Our prototype IPC system is used as a library by applications running in the 5620 environment. The library interface consists of four functions. Two of these are used to replace standard 5620 functions for monitoring resources.

```

void ipc_init(generic_id)
int generic_id;

```

The *ipc_init()* procedure initializes the IPC system and logs in the current process. The first process that attempts to initialize the system will spawn the IPC process, which in turn will load the IPC code. The IPC code remains resident, taking up a proc table entry, until the 5620 is rebooted. All subsequent processes using the system will share the IPC text space.

The *generic_id* may be used by other processes as a server name for communication with this process. More than one process may log in with the same generic id. Of course, messages may also be sent to a specific process referenced by the proc table pointer.

```
int ipc_wait(resource_mask)
    int resource_mask;
```

```
int ipc_own()
```

These functions are used to replace the resource monitoring functions in the 5620. `ipc_wait()` may be used to suspend the current process until one of the referenced resources is available. Supported resources include the keyboard, mouse, character queue from the host, time out alarm, and the incoming message queue. `ipc_wait()` and `ipc_own()` return a mask indicating the resources that are currently available.

```
message ipc_message (type, tag, proc, generic_id, data)
    int         type;
    int         tag;
    struct Proc *proc;
    int         generic_id;
    msdata      *data;
```

`ipc_message()` is the function that does most of the work in the IPC system. Error conditions are indicated by negative values in the type field of the returned message. The particular operation is selected by the type field from those listed below.

```
LOGIN
LOGOUT
SENDMSG
RCVMSG
RELEASEMSG
NAMESERVICE
```

LOGIN and LOGOUT set up or free the message queue structure for the current process. (In practice these are seldom used.) `ipc_init` calls the login function, and the message queue is automatically freed when the process exits.

SENDMSG is used to send a message to a generic or specific process, with the specified tag and message data. RCVMSG returns the first (fifo order) message in the queue or the first message from the specified process with the specified tag. RELEASEMSG is used to inform the system that the message is freeable.

NAMESERVICE has two uses. First it may be used to determine if a process is resident. Second it may be used to peruse the message queue structure of a process. This second use allows a process to monitor the system.

2.4 Applications

We have applied this IPC system to two classes of research problems that particularly interest us — systems that share data in a graphics environment and systems composed of several processes that behave as one application to the end user. An example of the former case is a system in which processes share fonts in a remote graphics environment. Shared data flows from a font caching process to each of several graphics processes in the

environment.

The latter problem is illustrated by a multi-representation file manager [5] in which communication between four modules is mediated by the IPC system. Any action taken in one module, or process, (one window per process) sends appropriate messages to the others to maintain a consistent, visible state of the working environment.

A simplified code fragment, based on the file management application, demonstrates the use of our IPC system. The graph module, *graphmod* waits for a message or some other event, and then attempts to read and release a message from the *dirmod* (directory editor) server. Slightly more detailed examples may be found in Appendix B.

```
main()
{
    int got;
    message m;

    request(KBD|MOUSE|RCV);
    ipc_init(GRAPHMOD);

    for(;;)
    {
        got = ipc_wait(KBD|MOUSE|RCV|IPCMMSG);
        if(got & IPCMSG)
        {
            m = ipc_message(RCVMSG, 0, 0, DIRMOD, 0);
            /* No messages from DIRMOD */
            if (m.type < 0)
                continue;
            /* code to handle the messages */
            ipc_message(RELEASEMSG, 0, 0, 0, &m);
        }
    }
}
```

Figure 4. Sample IPC Code

3. Comparisons with Host Level IPC

3.1 IPC Systems compared

We compared our prototype system with two commonly used IPC facilities under UNIX — System V (Release 2) and Berkeley sockets — as well as with Presotto and Ritchie's Ninth Edition IPC [14]. Since the IPC systems differ in so many ways, with ours having many more application-oriented features, this is at best an apples-and-oranges sort of comparison. In all fairness, our IPC was written for a prototype system and makes no claims to being well designed or any of the other mom-and-apple-pie things a good system should emulate. The best things it has going for it are that it works and does what we want.

System V IPC allows a limited form of message selection by type, which is not as flexible as our system. The messages are of variable length without any internal structure, i.e., just a lump of memory. This allows the application to send a structure or an array. However, arbitrary-length linked lists are a bit harder to implement with this scheme.

Berkeley sockets IPC seems to be just the antithesis of our system. A lot of time was spent designing the system, and it is fairly difficult to use. The sockets code is longer than both ours and the System V code by at least 20%. The socket reliable transmission model is similar in function to pipes, a structureless character stream. This means that for sending variable length messages, two *reads* have to be issued or some application supplied buffering scheme has to be used. In Research Versions 8 and 9, however, record size is preserved for writes and subsequent reads, allowing efficient implementation of variable-length messages.

One difficulty in comparing our prototype with the various host-based IPC systems is that the host vs. 5620 distinction is confounded with the particular prototype system design. To get a rough idea of how well our prototype would run on the host, we implemented the basic model as a pseudo-device driver, under UNIX Version 9 on a MicroVAX-II. Although similar in essence to the 5620-based IPC system, this host-based prototype has some limitations and somewhat reduced functionality. It will be referred to as the V9 prototype.

Like the 5620-based version, the V9 prototype allows processes to send and receive variable-sized messages. However, it requires two copy operations — one from the sending process to kernel space, and the other from kernel space to the receiving process — as opposed to the single copy in the 5620-based prototype. Some other differences are that the message data structure is just a flat hunk of memory rather than a null terminated linked list, there is only one queue for each process, and two processes with the same logical id cannot be logged in at the same time. Although the V9 prototype has been tested only on Version 9, portability to other UNIX versions should be straightforward.

3.2 Machines and benchmarks

The machines tested were the 5620 for our IPC system, the UNIX PC [3] and 3B2/400 for System V IPC, the Digital Equipment Corporation MicroVAX II for UNIX Version 9, and the Sun3/75 for the Berkeley 4.2 socket IPC system. We included two System V machines mainly because they were easy to acquire in our environment.

Since each IPC system runs on its own computing system and hardware, such differences have to be factored out to make meaningful comparisons among alternative IPC approaches. The benchmark chosen to get a raw measure of machine performance was the *sieve of erosthenes*. The code, which is identical for all machines, appears in Appendix A.

Benchmark timings are shown in Table 1. Time values, accurate to the nearest second, were taken from calls to `time(2)` in the case of UNIX-based code and a time emulator for the 5620. Each case presented here was run many times; time values were tested against other known sources and shown to be accurate.

The top row of Table 1 indicates the absolute performance of each machine / IPC system. According to this benchmark, the SUN3 is the fastest, while the 5620 is the lowest horsepower vehicle — a factor of about 4 to 1. Relative (to 5620) performance, shown in the bottom row, was obtained by dividing 29 (the *sieve* result for the 5620) by each value in

the top row.

Measure	5620	UNIX PC	3B2/400	MicroVAX II	Sun3/75
Sieve (seconds)	29	22	13	14	7
Relative Performance	1.0	1.3	2.2	2.1	4.1

TABLE 1. Relative Performance of the Machines Tested

Raw IPC performance was then obtained by sending 10 K identical messages from one process to another. The code for testing our system, a *send/receive* pair of programs, is included in Appendix B.

Measure	5620 Proto	UNIX PC SV.2	3B2/400 SV.2	MicroVAX II V9	MicroVAX II V9-proto	Sun3/75 BSD4.2
msgs/second	296.8	409.5	372.4	393.8	365.7	386.4
Relative msgs/second	296.8	310.7	166.9	190.1	176.6	93.3

TABLE 2. Relative IPC Performance For 10K Variable Length Messages Sent and Received

The top row of Table 2 gives the messages-per-second results obtained from the respective IPC benchmark programs. We reasoned that these values could be divided by relative machine performance (bottom row of Table 1) to get a machine-independent performance estimate for each IPC system. Although this normalization may not be optimal, it does allow for some rough comparisons among IPC systems on different machines. Results based on such relative messages per second appear in the bottom row of the table.

According to the relative measure, the two top performers by a reasonable margin are System V.2 running on the UNIX PC and our IPC system running in the 5620. Although the SUN3 itself is the fastest for the *sieve* benchmark, sockets-based IPC on the SUN3 fares worst in message sending efficiency when adjusted for machine speed. We feel that most of the differences among scores can be attributed to context switch / break point trap overhead. Our system has no break point overhead, while the SUN3 / BSD 4.2 requires two

kernel calls per variable-length message.

A surprising result is the better relative performance of the UNIX PC implementation of System V IPC in comparison with the 3B2/400. Although normalizing messages-per second by the *sieve* benchmark may not be the optimal adjustment, the substantial advantage of the UNIX PC is hard to explain.

The performance of the Ninth edition prototype is in the same ball park as the other host-based systems. With total functionality, performance would be somewhat reduced, however. In any case, the penalty for the enhanced features of our IPC approach is small for the host-based (Version 9) prototype, and eliminated when the system runs directly in the graphics engine.

3.3 Conclusions

Our IPC prototype is both easy to use and a rich environment in which to produce multiprocess bitmap graphics applications. Using programs as piece parts connected by IPC and working together as a single application appears to offer real advantages. Quality should be improved since programs tend to be more robust than subroutines, and productivity gains may come from code reuse.

Designing the IPC to run in the bitmap engine rather than in a more complex timesharing host computer also seems to be an advantage. The IPC system can be much faster running in such a low overhead operating environment, and this improvement in performance can be leveraged to improve the functionality and usability of the system.

References

- [1] —, "NeWS Preliminary Technical Overview," Sun Microsystems, Inc., October 2, 1986.
- [2] —, Note describing the version nine window system X, Massachusetts Institute of Technology, 1985.
- [3] —, "AT&T UNIX PC User's/Programmer's Manual," AT&T Information Systems, 1985.
- [4] Birrel, A.D., and Nelson, B.J., "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, vol. 2, no. 1, pp. 39-59, February 1984.
- [5] Blewett, C.D., Edmark, J.T., Helfman, J.I., and Wish, M., "A Multi-Representation, Bitmap Interface to the UNIX[®] File System Constructed from Cooperating Processes", Proceedings of Third Computer Graphics Workshop, USENIX, pp. 41-48, 1986.
- [6] Blewett, C.D., Freed, A., Hilbert, R.J., Langer, J., Mascitti, R.J., Rodine, C.R., and Weber, W.P., "The Aegis System," Proceedings of Second Computer Graphics Workshop, USENIX, 1986.
- [7] Cheriton, D.R., and Zwaenepoel, W., "The Distributed V kernel and its performance for diskless workstations," Proceedings of the 9th Symposium on Operating System Principles, ACM, 1983.
- [8] Gehani, N.H., and Roome, W.D., "Concurrent C", Software Practice and Experience, v16, no.9, pp. 821-844, 1986.
- [9] Goldberg, A.J., and Robson, D., "Smalltalk-80: The Language and Its Implementation," Addison-Wesley Publishing Company, Reading, MA, 1983.
- [10] Hoare, C.A.R., "Communicating Sequential Processes," CACM, v21, no. 8, pp. 666-677, August 1978.
- [11] Jones, J.B., and Rashid, R.F., "Mach and Matchmaker: Kernel and Language Support for Distributed Object-Oriented Systems," OOPSLA Conference Proceedings, 1986.
- [12] Leffler, S.J., Fabry, R.S., and Joy, W.N., "A 4.2bsd Interprocess Communication Primer, Draft of July 27, 1983," Report no. UCB/CSB 83/145, July 1983, Computer Science Division (EECS), University of California, Berkeley, California.
- [13] Pike, R., "The Blit: A Multiplexed Graphics Terminal," AT&T Bell Labs Tech. J. 63, #8, part 2, pp. 1607-1632, 1984.
- [14] Presotto, D.L., and Ritchie, D.M., "Interprocess Communication in the Eighth Edition Unix System," USENIX Summer Conference Proceedings, 1985.
- [15] Stroustrup, B., "A Set of C++ Classes for Co-routine Style Programming," AT&T Computing Science Technical Report, November 1, 1984.

Appendix A: Sieve of Eratosthenes Benchmark

```
#ifdef 5620
#include <jerq.h>
#include <font.h>
#else
#include <stdio.h>
#endif

#define SIZE 8191
char flags[SIZE];

main()
{
    register unsigned int i;
    register unsigned int prime;
    register unsigned int k;
    register unsigned int count;
    register unsigned int iter;
    unsigned long t1, t2;

    time (&t1);

    for (iter = 1; iter <= 100; iter++)
    {
        count = 0;
        for (i = 0; i < SIZE; i++)
            flags[i] = 1;
        for (i = 0; i < SIZE; i++)
        {
            if (flags[i])
            {
                prime = i + i + 3;
                for (k = i + prime; k < SIZE; k += prime)
                    flags[k] = 0;
                count++;
            }
        }

        time (&t2);
    }

#ifdef 5620

    time (t)
    unsigned long *t;
    {
        *t = realtime () / 60;
    }

#endif /* 5620 */
```

Appendix B—1: 5620 Benchmark — Sending Process

```
#include <jerq.h>
#include <font.h>
#include "ipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  31415

main ()
{
    register int sent;
    message m;
    msdata ms;
    unsigned long t1, t2;

    time (&t1);

    ipc_init (MSG_QUEUE_ID + 1); /* some random value */

    ms.action = 0;
    ms.name = "a random message";
    ms.length = strlen(ms.name);
    ms.next = (msdata *) 0;

    sent = MSG_MSGS;
    while (sent)
    {
        if (ipc_message (SENDMSG, 0, 0, MSG_QUEUE_ID, &ms).type >= 0)
            sent--;
        else
            wait (CPU);
    }

    time (&t2);
}

time (t)
unsigned long *t;
{
    *t = realtime () / 60;
}
```

Appendix B—2: 5620 Benchmark — Receiving Process

```
#include <jerq.h>
#include <font.h>
#include "ipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  31415

main ()
{
    register int rcvd;
    register int msgid;
    message m;
    msdata ms;
    unsigned long t1, t2;

    time (&t1);

    ipc_init (MSG_QUEUE_ID);

    rcvd = MSG_MSGS;
    while (rcvd)
    {
        if ((m = ipc_message (RCVMSG, 0, 0, 0, 0), m.type) != NOPE)
        {
            rcvd--;
            ipc_message (RELEASEMSG, 0, 0, 0, &m);
        }
        else
            ipc_wait (IPCMSG);
    }

    ipc_message (LOGOUT, 0, 0, MSG_QUEUE_ID, 0);

    time (&t2);
}
```


Appendix C—1: System V Benchmark — Sending Process

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSG_MSGS      (1024 * 10)
#define MSG_SIZE      64
#define MSG_QUEUE_ID  31415
#define READ_WRITE_ALL 0666
#define MAX_PRIORITY   1L

main ()
{
    register int sent;
    register int msgid;
    register int msgsz;
    struct mymsgbuf
    {
        long mtype;
        char data[MSG_SIZE];
    } msgb;
    unsigned long t1, t2;

    time (&t1);

    while ((msgid = msgget (MSG_QUEUE_ID, READ_WRITE_ALL)) == -1)
    {
        fprintf (stderr, "Cannot access the message queue\n");
    }

    msgb.mtype = MAX_PRIORITY;
    strcpy (msgb.data, "a random message");
    msgsz = strlen (msgb.data);

    sent = MSG_MSGS;
    while (sent)
        if (msgsnd (msgid, &msgb, msgsz, 0) != -1)
            sent--;

    time (&t2);
}
```

Appendix C—2: System V Benchmark — Receiving Process

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSG_MSGS      (1024 * 10)
#define MSG_SIZE      64
#define MSG_QUEUE_ID  31415
#define READ_WRITE_ALL 0666
#define MAX_PRIORITY   1L

main ()
{
    register int rcvd;
    register int msgid;
    register long mtype = MAX_PRIORITY;
    struct mymsgbuf
    {
        long mtype;
        char data[MSG_SIZE];
    } msgb;
    unsigned long t1, t2;

    time (&t1);

    if ((msgid = msgget (MSG_QUEUE_ID, IPC_CREAT | READ_WRITE_ALL)) == -1)
    {
        fprintf (stderr, "Cannot create the message queue\n");
        exit (0);
    }

    rcvd = MSG_MSGS;
    while (rcvd)
        if (msgrcv (msgid, &msgb, MSG_SIZE, mtype, 0) != -1)
            rcvd--;

    msgctl (msgid, IPC_RMID, &msgb);

    time (&t2);
}
```

Appendix D—1: Version 9 Benchmark — Sending Process

```
#include <stdio.h>
#include "ipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_SIZE      64

struct msgdata
{
    int size;
    char data[MSG_SIZE];
};

main ()
{
    register int nfd;
    register ipcinfo *ip;
    register int fd;
    register int sent;
    int size;
    struct msgdata msgb;
    unsigned long t1, t2;

    time (&t1);

    if ((fd = ipccreat ("/tmp/fm1", 0)) < 0)
    {
        fprintf (stderr, "cannot announce fm: %s\n", strerror);
        exit (1);
    }

    strcpy (msgb.data, "a random message");
    msgb.size = strlen (msgb.data);
    size = msgb.size + sizeof(int);

    if (ip = ipclisten (fd))
    {
        nfd = ipcaccept (ip);
        sent = MSG_MSGS;
        while (sent)
        {
            if (write (nfd, &msgb, size) == size)
                sent--;
        }
        close (nfd);
    }

    time (&t2);
}
```

Appendix D—2: Version 9 Benchmark — Receiving Process

```
#include <stdio.h>
#include "ipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_SIZE      64

struct msgdata
{
    int size;
    char data[MSG_SIZE];
};

#define TRUE          1
#define FALSE         0

main ()
{
    register int fd;
    register int rcvd;
    struct msgdata msgb;
    unsigned long t1, t2;

    time (&t1);

    while ((fd = ipcopen ("/tmp/fm1", "heavy")) < 0)
    {
        fprintf (stderr, "cannot connect to fm: %s\n", strerror);
    }

    rcvd = MSG_MSGS;
    while (rcvd)
    {
        if (read (fd, &msgb, sizeof(msgb)) > 0)
            rcvd--;
    }

    close (fd);

    time (&t2);
}
```

Appendix E—1: Sun3 4.2 Benchmark — Sending Process

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/un.h>

#define NUMBERINQUEUE 5

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  "/tmp/31415"

main ()
{
    register int sent;
    register int s;          /* socket descriptor */
    int len;
    register char *msg;
    struct sockaddr_un server;
    unsigned long t1, t2;

    time (&t1);

    if ((s = socket (AF_UNIX, SOCK_STREAM, 0)) == -1)
    {
        fprintf (stderr, "cannot create the socket\n");
        exit (0);
    }

    server.sun_family = AF_UNIX;
    strcpy (server.sun_path, MSG_QUEUE_ID);

    if (connect (s, &server, strlen(server.sun_path) + 2) == -1)
    {
        fprintf (stderr, "cannot connect to socket\n");
        exit (0);
    }

    msg = "a random message";
    len = strlen (msg);

    sent = MSG_MSGS;
    while (sent)
    {
        if (send (s, &len, sizeof (int), 0) == -1)
            continue;
        if (send (s, msg, len, 0) != -1)
            sent--;
    }

    time (&t2);
}
```


Appendix E—2: Sun3 4.2 Benchmark — Receiving Process

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/un.h>

#define NUMBERINQUEUE 5

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  "/tmp/31415"

main ()
{
    register int rcvd;
    register int fd;
    register int len;
    register int s;
    int fromlen;
    char *file = MSG_QUEUE_ID;
    char msg[BUFSIZ];
    struct sockaddr_un sbox;
    struct sockaddr from;
    unsigned long t1, t2;

    time (&t1);

    if ((s = socket (AF_UNIX, SOCK_STREAM, 0)) == -1)
    {
        fprintf (stderr, "cannot create the socket\n");
        exit (0);
    }

    sbox.sun_family = AF_UNIX;
    strcpy (sbox.sun_path, file);
    if (bind (s, &sbox, strlen(sbox.sun_path) + 2) == -1)
    {
        fprintf (stderr, "cannot bind the socket name\n");
        exit (0);
    }

    while (listen (s, NUMBERINQUEUE) == -1)
        fprintf (stderr, "some error occurred while listening\n");

    fromlen = sizeof (from);
    if ((fd = accept (s, &from, &fromlen)) == -1)
    {
        fprintf (stderr, "error on accept\n");
        exit (0);
    }

    rcvd = MSG_MSGS;
    while (rcvd)
    {
        if (recv (fd, &fromlen, sizeof (int), 0) != sizeof (int))
```

```

        continue;
    if ((len = recv (fd, msg, fromlen, 0)) == fromlen)
        rcvd--;
}
close (fd);

shutdown (s, 2);
close (s);
unlink (file);

time (&t2);
}

```

Appendix F—1: Version 9 Host-Based Prototype Benchmark — Sending Process

```
#include <stdio.h>
#include "jipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  31415+1

static message blankmessage;
#define CLEAR_MESS(m)  (*m) = blankmessage;

main ()
{
    register int sent;
    register message *m;
    char mbuf[sizeof(message) + DATASIZE + 1];
    int fd;
    unsigned long t1, t2;

    time (&t1);

    if((fd = open("/dev/jipc",0)) == -1)
    {
        fprintf (stderr, "open failed\n");
        exit(1);
    }
    m = (message *) mbuf;
    CLEAR_MESS(m);
    m->id = MSG_QUEUE_ID;

    if (ioctl (fd, J_LOGIN, m) == -1)
    {
        fprintf (stderr, "Cannot login to ipc\0);
        exit (1);
    }

    sent = MSG_MSGS;
    strcpy (m->data, "a random message");
    m->size = strlen (m->data);
    while (sent)
    {
        m->code = J_WAIT;
        m->tag = 0;
        m->id = 0;
        m->procid = 0;
        m->id = MSG_QUEUE_ID + 1;

        if (ioctl (fd, J_SENDMSG, m) >= 0)
            sent--;
    }

    for (;;)
    {
        CLEAR_MESS(m);
        m->code = J_WAIT;
        m->id = MSG_QUEUE_ID + 1;
```

```

        if(ioctl(fd, J_STATUS, m) == -1)
            break;
        sleep(1);
    }
    close (fd);

    time (&t2);
}

```

Appendix F—2: Version 9 Host-Based Prototype Benchmark — Receiving Process

```
#include <stdio.h>
#include "jipc.h"

#define MSG_MSGS      (1024 * 10)
#define MSG_QUEUE_ID  31415+1

static message blankmessage;
#define CLEAR_MESS(m)  (*m) = blankmessage;

main ()
{
    register int rcvd;
    register message *m;
    char mbuf[sizeof(message) + DATASIZE + 1];
    int fd;
    unsigned long t1, t2;

    time (&t1);
    if((fd = open("/dev/jipc",0)) == -1)
    {
        fprintf (stderr, "open failed\n");
        exit(1);
    }

    m = (message *) mbuf;
    CLEAR_MESS (m);
    m->id = MSG_QUEUE_ID + 1;

    if (ioctl (fd, J_LOGIN, m) == -1)
    {
        fprintf (stderr, "Cannot login to ipc\n");
        exit (1);
    }

    rcvd = MSG_MSGS;
    while (rcvd)
    {
        CLEAR_MESS (m);
        m->size = DATASIZE;
        m->id = MSG_QUEUE_ID;
        m->code = J_WAIT;

        if (ioctl (fd, J_RCVMSG, m) >= 0)
            rcvd--;
    }

    close (fd);

    time (&t2);
}
```


Mach Threads and the Unix Kernel: The Battle for Control

Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub,
David L. Black, Eric Cooper and Michael W. Young.

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This paper examines a kernel implemented lightweight process mechanism built for the Mach operating system. The pros and cons of such a mechanism are discussed along with the problems encountered during its implementation.

1. Introduction

The early Unix notion of process was based on the hardware abstraction of its day: a single CPU executing within a memory address space. Even today, although, multiprocessors are becoming increasingly common, neither Unix System V nor 4.3 BSD provide a way to manage more than one thread of control within an address space.

The addition of lightweight processes to Unix would provide many advantages. In fact, the lack of kernel support has caused Unix programmers to implement a variety of coroutine packages to support multi-stack applications. Lightweight threads of control can allow a programmer to encapsulate computations with their stack state and thus achieve greater modularity. Research systems, such as THOTH [2] and its successor, Stanford's V Kernel [3], have shown that multiple threads of control within a single process can be an especially important tool for writing server applications. A thread package could provide an attractive way to take advantage of the parallelism afforded by tightly-coupled shared memory multiprocessors.

This paper examines a kernel-implemented thread facility built for the Mach operating system [1]. The pros and cons of such a mechanism are discussed along with the problems encountered during its implementation.

2. Kernel Implemented Threads vs. Coroutines

Some of the advantages of lightweight processes can be achieved by out-of-kernel solutions, but often at the expense of either preemption or parallelism. Two approaches are common: multi-process shared memory implementations and single process coroutines.

Parallel execution can be achieved with multiple processes in conjunction with some kind of shared memory facility. For example, in Dynix [4] users can allocate a number of processes equal to the number of processors and effectively manage shared computations through an *mmaped* region of shared memory. Similar tricks allow programmers to build multiprocessor applications using Encore's UMAX [5] operating system. Typically such systems amount to a second layer of scheduling similar to that used within the operating system itself.

A significant advantage of coroutine packages is that they can significantly reduce the costs of multi-thread management or at least isolate them within a user process. Out-of-kernel coroutine packages do, however, have many problems:

- Scheduling is very difficult to do. Most coroutine packages use non-preemptive scheduling.
- It is impossible to have truly parallel execution in a pure coroutine package.
- If a coroutine takes a page fault or other type of trap that causes it to wait (e.g. for disk I/O), then all other coroutines must wait.
- Only a single coroutine may be executing a system call. Therefore, if a coroutine executes a system call that blocks causes the entire set of coroutines to block.

The primary disadvantage of an in-kernel thread implementation is potential cost. In addition to the cost of crossing the user process/kernel protection boundary with a trap or system call, there is also the cost of thread data structures, which must be managed in kernel virtual address space, and the cost of general purpose preemptive scheduling, which will typically be much higher than those of a specialized coroutine package.

After considering the alternatives and their problems, it was decided that in Mach it would make sense to provide primitive multi-threaded support within the kernel which would provide for both parallelism and preemption. This support would then serve as the base upon which lightweight process packages could be implemented.

3. Mach Task and Thread Primitives

Mach splits the Unix abstraction of process into two components: the *task* and the *thread*. A Mach *task* consists of a collection of system resources, including an address space. It can be thought of as that part of a Unix process consisting of its address space, file descriptors, resource usage information, etc. In essence, a task is a process without a flow of control or register set (hardware state).

A Mach *thread* is the basic unit of execution. A thread executes within the context of exactly

one task. However, any number of threads may execute within a single task. Threads execute in pseudo-parallel on a uniprocessor. When running on a tightly coupled multiprocessor, multiple threads may execute in parallel. A traditional BSD process is implemented in Mach as a task with a single thread of control.

Appendices I and II list the operations supplied by Mach for creating, managing and destroying tasks and threads. Note that all such operations are performed using object handles which are in fact capabilities to communication channels (i.e., Mach *ports*).

4. User Level Thread Synchronization

At any given point in time, a thread can be in one of three states:

1. A thread that is in *running state* is either executing on some processor, or is eligible for execution on a processor as far as the user is concerned. A thread may be in running state yet blocked for some reason inside the kernel (perhaps waiting for a page fault to be handled).
2. If a thread is in *will-suspend* state, then it can still execute on some processor until a call to *thread_wait* is invoked.
3. A thread that is in *suspended* state is not executing on processor. The thread will not execute on any processor until it returns to running state.

Each of these states can also apply to a task. That is, a task may be in running, will-suspend or suspended state. The state of a task will affect all threads executing within that task. For example, a thread can be eligible for execution only if both it and its task are in the running or the will-suspend state.

The Mach kernel does not enforce a synchronization model. Instead, it provides basic primitives upon which different models of synchronization may be built. One form such synchronization could take would be the Mach IPC facility [1]. Should an application desire its own thread-level synchronization, it can use the suspend, resume and wait primitives. For example, to implement P and V style semaphores with shared memory, one could use:

```

P(semaphore)
{
    lock(semaphore);
    while (semaphore->inuse) {
        thread_suspend(thread_self());
        enqueue(semaphore->queue, thread_self());
        unlock(semaphore);
        thread_wait(thread_self(), TRUE);
        lock(semaphore);
    }
    semaphore->inuse = TRUE;
    unlock(semaphore);
}

V(semaphore)
{
    lock(semaphore);
    semaphore->inuse = FALSE;
    next = dequeue(semaphore->queue);
    if (next != THREAD_NULL)
        thread_resume(next);
    unlock(semaphore);
}

```

In this example, lock and unlock could be implemented as spin locks on shared memory. To perform the P operation, the caller checks if the semaphore is in use. If so, it puts itself on a queue of threads waiting for the semaphore and goes into a suspended state. When it is placed back in running state it once again checks the semaphore, suspending itself again if necessary. To perform the V operation, a thread checks for other waiting threads and places the first thread in the semaphore queue into the running state.

Note that placing a thread into the suspended state is separated into a suspend operation followed by a wait operation. If there were not such a separation, it would be impossible for an application to correctly synchronize unless the kernel provided semaphores directly. If suspend/wait were a single operation a thread would be forced to call it either before or after unlocking the semaphore. If the thread made the call before unlocking the semaphore then the application would deadlock because the semaphore was never unlocked. If the thread made the call after unlocking the semaphore then it would be possible for the thread holding the semaphore to perform its resume before the waiting thread is able to suspend itself. In this case, the thread would suspend itself and never be resumed.

Of course, the kernel could implement semaphores directly (as does, for example, System V). It was felt, however, that a semaphore package would only add yet another synchronization mechanism to the kernel on top of that provided by the Mach IPC facility. The kernel would inevitably implement only a small set of semaphore types and applications that wanted to use different semaphore semantics would still be forced to use an extra layer of synchronization and manage additional data structures.

5. The C-Threads Package

The exported thread primitives are intentionally low level to allow flexibility in dealing with a variety of programming languages and architectures. By providing a minimal kernel interface, it is possible to implement many different application or language interfaces to threads without burdening some applications in favor of others. For example, a feature that provided dynamically growing stacks might be useful for a naive C programmer, but it might be extra baggage for a Lisp programmer.

Higher level interfaces to threads can be provided in the form of:

- run time libraries,
- new language constructs and/or
- home grown packages developed for specific applications.

One such high-level package for programming in C, called *C-threads*, has already been implemented. It provides a high level C interface to the low level thread primitives along with a collection of other mechanisms useful in various parallel programming paradigms (similar to those available in languages such as Mesa [6]).

The C-Threads package provides

- multiple threads of control for parallelism,
- shared variables,
- mutual exclusion for critical sections and
- condition variables for synchronization of threads.

To provide multiple threads of control, the C-Threads interface defines *cthread_fork* for creating new threads, *cthread_exit* for exiting threads and *cthread_join* to wait for a particular thread to finish.

Threads that wish to access shared data may use the mutual exclusion facilities provided by C-Threads. In particular, *mutex_alloc* and *mutex_free* allocate and deallocate *mutex* objects. The *mutex* objects support the functions *mutex_lock* and *mutex_unlock* which correspond to typical P and V operations.

Synchronization in C-Threads may also be accomplished with condition variables. *Condition_alloc* and *condition_free* allocate and free condition variables. When a thread wishes to indicate that a condition is true, it uses *condition_signal* to awaken at least one of the threads waiting for the condition. The *condition_broadcast* primitive causes all threads waiting for a condition to wake up. A thread may of course wait for a condition using *condition_wait*.

There are, currently, three separate C-Threads implementations. The first implements threads as coroutines in a single task. The second uses a separate task for each *cthread*, using inherited shared memory to partially simulate the environment in which multiple threads run. The third

implementation uses the thread primitives provided by the kernel.

The coroutine version is generally easier to use for debugging since the order of context switching is repeatable and the user need not worry about concurrent calls to C library routines. However, the coroutine version can not exhibit parallelism as the other two versions do. The multiple task version can be an effective way to achieve parallelism on architectures which do not allow full, uniform access-delay sharing of memory.

6. The Effect of Threads on Unix Features

From a Unix programmer's perspective, the separation of the Unix protection domain from its control abstraction has been accomplished at no apparent cost. Mach provides complete emulation of 4.3BSD Unix, even for binaries on VAX machines. Overall system performance has not been eroded.

However, should one desire to use a multithreaded task along with Unix features, there are many potential pitfalls. Unix was not designed to work in a multithreaded environment. Some of the obvious problems are:

- The semantics of common functions (e.g. `fork`) are not well defined in the presence of multiple threads.
- Many standard library routines return results in static areas.
- Most C compilers return structures in a static area.
- The definitions of static returned values such as `errno` are inappropriate for a multithreaded environment.
- Many library routines, never expected to be run in a multithreaded application, are coded in non-reentrant ways. Many traditional Unix libraries would not even work if a signal routine were to be called at the wrong time!

Where the semantics of Unix operations are not well defined in the presence of multiple threads, it was necessary to determine some reasonable definition. Two examples of this are *fork* and *signals*.

The Unix `fork` primitive raises the question: "When a thread in a task containing multiple threads executes the `fork` system call, which threads does the child task contain?". There are two possible answers:

1. The child task contains exactly one thread corresponding to the calling thread.
2. The child task contains the same number of threads as the parent. Each thread in the child corresponds to a thread in the parent.

Mach implements the first choice, which is really the most logical when the properties of tasks and threads are considered. `Fork` is largely an address space manipulation and corresponds very closely to the `task_create` operation. Unix process semantics dictate that the child must contain at least one thread. The logical choice for this thread is a replica of the calling thread. This choice also corresponds to the common case when a thread within a server task decides to `fork` to

perform an operation in a separate address space.

Signals present an interesting problem in the domain of multiple threads. Are signals sent to tasks or threads? Considering that the logical equivalent of a Mach task is a Unix process, and that signals are sent to processes, it is appropriate to define signals as being sent to a task. Unfortunately, a task is not an executable unit and can therefore not handle a signal. To overcome this problem, the Mach kernel chooses some thread within the task to handle the signal. The actual thread that will handle the signal is not well-defined. In fact, the current implementation causes the first thread to notice the signal to be the handler. This is clearly not an optimal solution because it can seriously confuse a Unix programmer that wishes to use signals to cause a stack unwinding operation such as *longjmp*. The better long term solution is to convert signals into Mach IPC messages. Each task could then designate one or more threads that would receive signals on a special *signal port*.

7. Implementation: Details, Problems and Issues

Within the Mach kernel, the task (sic) of incorporating threads exacted a significant toll on the implementors. This was due to the fact that Mach currently provides for Unix compatibility directly in the kernel. The 4.3 BSD kernel code was designed (presumably unintentionally) to make a multiple thread per address space implementation very difficult. For example, both the u-area and kernel stack reside in the user's address space and are even assumed to exist at the same address for all processes. Unix process management is not restricted to a handful of scheduling modules. Instead, it is spread throughout unrelated kernel code. A form of process management can even be found in device drivers. The 4.3 BSD signal mechanism is neither well defined nor even appropriate for such an environment.

Perhaps the most annoying problem was that of u-area management. There are literally thousands of lines of kernel code that use *u.* to reference the u-area directly. This assumes that the u-area is at the same address for all processes. Since threads must share an entire address space and must each have their own u-like data structure, the traditional u-area cannot exist at a single, unique address. In fact, the problem is even worse: some fields of the old u-area refer to data which should be thread specific properties while other fields refer to task specific properties. Therefore, within the BSD compatibility code, each u-area reference can no longer be a simple memory reference to a fixed address. Instead, each u-area reference must be a pointer dereference with the pointer depending on whether the desired field is a task or thread feature. Rather than inspect and modify each of the thousands of lines of C code, a few tricks were played with the C preprocessor. Two new structures, *uthread* and *utask* were defined to hold the thread and task specific u-area information. For example:

```

struct uthread {
    int    uu_thread1;
    int    uu_thread2;
    .
    .
    .
};

struct utask {
    int    uu_task1;
    int    uu_task2;
    .
    .
    .
};

```

uu_thread1, uu_thread2, ... corresponded to fields in the typical Unix u-area. Next, *u* itself was defined as follows:

```
#define u      (current_thread()->u_address)
```

with the *u_address* field of the thread structure defined as:

```

struct thread {
    .
    .
    .
    struct u_address {
        struct uthread *uthread;
        struct utask  *utask;
    } u_address;
    .
    .
    .
};

```

Finally, each potential u-area field was defined as:

```

#define u_thread1    uthread->uu_thread1
#define u_thread2    uthread->uu_thread2
.
.
.
#define u_task1      utask->uu_task1
#define u_task2      utask->uu_task2

```

When a task is created it is allocated a utask structure. When a thread is created it is allocated a uthread structure. The pointer to this structure, along with the pointer to the task's utask structure, are then saved in the *u_address* sub-structure of the thread structure.

The good news is that these definitions handle almost all uses of the u-area. The bad news is that most u-area references change from one instruction to several. This increases both execution time and code space. After some intense hacking, each u-area reference was reduced to only 3

VAX instructions. The first instruction fetches the current thread, the second instruction loads the appropriate pointer (uthread or utask), and the third instruction performs the actual u-area reference.

Even though each u-area reference is now significantly more expensive than in a standard Unix system, the Mach kernel still performs better than a 4.3 kernel in measurements of overall performance -- largely due to improvements in Mach's handling of virtual memory. For example, to compile all programs in /bin on a vanilla 4.3 system (using a CMU enhanced cc and cpp) takes 1017 seconds on a VAX 780 (with a Fujitsu Eagle disk drive). A Mach kernel without multiple thread support and normal u-area references takes only 964 seconds. A Mach kernel with multiple thread support and the expensive u-area references requires 986 seconds to complete the test. Given that approximately 700 seconds is spent in user time, and since a Mach kernel can not improve on the user time of existing binaries, it makes sense to factor that 700 seconds out of the measurements. Therefore, we see that 4.3 is responsible for $1017 - 700 = 317$ seconds. A non-thread Mach kernel is responsible for $964 - 700 = 264$ seconds. A thread Mach kernel is responsible for $986 - 700 = 286$ seconds. So, a non-thread Mach kernel is approximately $(317-264)/317 = 16.7\%$ faster than 4.3. A thread Mach kernel is still $(317-286)/317 = 9.7\%$ faster than 4.3.

It is expected that some improvement in the kernel supporting threads will be gained by identifying u-area hotspots: those places in the kernel that make many references to the u-area. Once identified, these sections of kernel code can be reworked to avoid using the u-area, or to use it in a more efficient way.

8. Performance Issues

While an order of magnitude less expensive than the Unix *fork*/*exit* operations, *thread_create* and *thread_terminate* are still moderately expensive operations as indicated in table 8-1. In addition to the O(1 millisecond) each operation takes on a MicroVAX II, there are also the memory costs incurred by the thread data structures themselves and the necessity for a (pagable) kernel stack for each thread which must be physically resident when the thread is runnable. For this reason, an application that needs huge numbers of thread-like entities (perhaps millions) would probably be best implemented as a hybrid of kernel-supplied threads and coroutines. That is, a huge number of coroutines would map to a much smaller number of threads executing in a single process. The number of threads used could correspond either to the number of available processors or the number of concurrently executing system calls or traps.

Some of the costs of threads can be "optimized away". For example, the C-threads package caches threads which have exited so they can be reused when a new *cthread_fork* is called. In a multiprocessor with a sufficient number of processors, context switching is eliminated entirely. On a single processor machine, however, many of the costs of scheduling Unix processes remain in the scheduling of threads.

Fork/Exit vs. Thread Create/Terminate	
Kernel operation	VAX Instructions Executed (typical case)
fork	3069
wait3	3440
exit	916
fork/wait3/exit (total)	7425
thread_create	375
thread_exit	409
thread_total	784

Table 8-1:
Number of VAX CPU instructions executed.
(Mach, MicroVAX II, 4K page size)

9. Conclusion and Status

The Mach thread implementation is running (April 1987) on multiprocessor and uniprocessor VAX, Encore and Sequent machines within CMU. A version of Eric Cooper's C-Threads package which uses threads is also working. Mach is being released externally to interested researchers. The first release (Release 0) of Mach began in December of 1986.

10. Acknowledgements

The multiple thread kernel support was implemented by Avie Tevanian, David Golub and David Black. Eric Cooper implemented the C-Thread package and along with others had much input on the final interface. No one in their right mind would claim credit for thinking up the u-area hacks.

I. Thread Operations

Following is a list of all kernel supported thread operations:

```
thread_create(task, child, child_data)
    task_t      task;          /* parent task */
    thread_t     *child;        /* new thread */
    port_t       *child_data;   /* child data port */
```

Thread_create create a new thread in the specified task. Initially, the thread is in *suspended* state and its registers contain undefined values.

```
thread_terminate(thread)
    thread_t     thread;        /* thread to terminate */
```

Thread_terminate destroys the specified thread.

```
thread_suspend(thread)
    thread_t     thread;        /* thread to suspend */
```

The specified thread is placed in *will-suspend* state.

```
thread_resume(thread)
    thread_t     thread;        /* thread to resume */
```

The specified thread is placed in *running* state.

```
thread_wait(thread, wait)
    thread_t     thread;        /* thread to cause to wait */
    boolean_t    wait;          /* wait for it to stop? */
```

If the specified thread is in *will-suspend* state then *thread_wait* will place it in *suspended* state. If the *wait* parameter is TRUE, the calling thread will wait for the thread to come to a complete stop.

```
thread_status(thread, status)
    thread_t     thread;        /* thread to query */
    thread_status_t *status;     /* thread status information */
```

Thread_status returns the register state of the specified thread. The *status* parameter returns the address of a machine-dependent status structure describing the register state for the machine type the thread is executing on.

```
thread_mutate(thread, status)
    thread_t     thread;        /* thread to mutate */
    thread_status_t *status;     /* status information to set */
```

Thread_mutate sets the register state of the specified thread. As in *thread_status*, the *status* structure is machine-dependent.

II. Task Operations

Following is a list of all kernel support task operations:

```
task_create(parent, inherit, child, child_port)
    task_t      parent;          /* the parent task */
    boolean_t   inherit;         /* pass VM to child? */
    task_t      *child;          /* new task */
    port_t      *child_port;     /* new task's data port */
```

Task_create creates a new task. The child's address space is created using the parents inheritance values if the *inherit* flag is TRUE. If the *inherit* flag is FALSE, the child is created with an empty address space. Access to the child's task and data ports are returned in *child* and *child_port* respectively.

```
task_terminate(task)
    task_t      task;           /* task to terminate */
```

The specified task is destroyed.

```
task_suspend(task)
    task_t      task;           /* task to suspend */
```

The specified task is placed in *will-suspend* state.

```
task_resume(task)
    task_t      task;           /* task to resume */
```

The specified task is placed in *running* state.

```
task_wait(task, wait)
    task_t      task;           /* task to cause to wait */
    boolean_t   wait;           /* wait for it to stop? */
```

If the specified task is in *will-suspend* state then *task_wait* places it in *suspended* state. If the *wait* flag is TRUE the calling thread will wait for all threads in the task to come to a complete stop.

```
task_threads(task, list)
    task_t      task;           /* task to generate list for */
    thread_t    *list[];        /* list of threads */
```

Task_threads returns the list of all threads in a task.

```
task_ports(task, list)
    task_t      task;           /* task to generate list for */
    port_t      *list[];        /* list of ports */
```

Task_ports returns the list of all ports the specified task has access to.

References

- [1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young.
Mach: A New Kernel Foundation for UNIX Development.
In *Proceedings of Summer Usenix*. July, 1986.
- [2] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM :105-115, February, 1979.
- [3] D. R. Cheriton and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proceedings of the 9th Symposium on Operating System Principles*, pages 128-139.
ACM, October, 1983.
- [4] Sequent Computer Systems, Inc.
Dynix Programmer's Manual
Sequent Computer Systems, Inc., 1986.
- [5] Encore Computer Corporation.
UMAX 4.2 Programmer's Reference Manual
Encore Computer Corporation, 1986.
- [6] Lampson, B.W. and D.D. Redell.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-113, February, 1980.

A Dynamically Extensible Streams Implementation

Jim Rees, Margaret Olson, J. Sasidhar

apollo computer inc
330 Billerica Road
Chelmsford, MA 01824

Introduction

System V Streams [1,2] provides a uniform execution environment for communications protocols in the Unix operating system [3]. Our implementation is based on the Domain system, and takes advantage of several features of that system to make it easier to write, debug, install and configure Streams modules and drivers.

Streams modules, buffers, and queues all reside in user global address space, and are accessible from any user process. They are not bound in with the operating system kernel. Modules and drivers are self-contained and self-describing. New protocols may be added to any configuration without rebuilding anything, and without any knowledge of the existing configuration.

Protocols are accessible from both the Streams defined interfaces and from existing Domain interfaces, including the Open System Toolkit [4]. This is made possible by a global Type Manager that manages all Streams protocol types.

Goals

We had several goals in mind for this project:

- Provide a consistent base for implementing communications protocols.
- Make it easy to port Streams modules and drivers from other systems.
- Make it easy to install and configure Streams modules and drivers.
- Provide performance at least as good as that of existing communications products implemented on the Domain system.

The Streams Environment

A *stream* consists of a *stream head*, a *driver*, and zero or more *modules*. When a stream is first opened, it connects a user process directly to the driver, which may be a real device driver, a pseudo-device driver, or a *multiplexor*. The stream head implements the familiar Unix I/O

Unix is a trademark of AT&T.

DOMAIN is a registered trademark of Apollo Computer Inc.

Copyright 1987 Apollo Computer Inc.

primitives, such as read and write. It transforms the procedural interface provided by the I/O system into the message passing interface understood by the modules. The modules each consist of two *queues*, providing for full-duplex processing in the *upstream* (toward the head) and *downstream* (toward the driver) directions. The modules communicate with the head, the driver, and each other by passing messages contained in *message blocks*. Modules are connected together in a stack, and may be pushed and popped on the stream by making an `ioctl` call.

A multiplexor is a type of device driver that may have several streams attached to it both on the upstream and downstream sides. Multiplexors are typically used to implement transport providers, such as TCP.

Although originally designed as a replacement for the `clist` structs used by the `tty` routines in the Unix kernel, Streams is currently used primarily as the foundation on which to build communications protocol packages. The protocols are implemented as multiplexors and modules, and are connected between client user processes and network devices. The complete stack of modules and multiplexors implementing a particular protocol is called a *protocol stack*.

In System V, the modules, drivers and message blocks are statically configured into the kernel. Adding a new module or driver, or changing the size of the message block pool, requires configuring a new kernel and rebooting. The modules and drivers run in a kernel context, so debugging requires the use of kernel debugging tools and skills. An errant module or driver may cause the system to crash. On those systems that do not support kernel paging, all of the code and data must be resident in physical memory, even if it is not used in a particular configuration.

Our implementation of Streams runs entirely in user space. Modules and drivers are loaded into user global address space at boot time, but are not part of the kernel. They are demand paged into virtual memory as they are needed. The standard user debugger can be used to operate on them. Kernel data structures are protected from faulty modules by the same kernel address space protection boundary that protects the kernel from all user space code.

The Unix System V Kernel Environment

Streams modules and drivers make certain assumptions about the environment in which they run that are not valid in a user space context. Validating these assumptions is one of the more difficult problems involved in moving the Streams system to user space.

Unix kernels are typically single threaded. System calls running on behalf of a user process run to completion without the possibility of another system call running at the same time, unless the first system call explicitly relinquishes use of the processor. This greatly reduces the need to put mutual exclusion locks on kernel data structures during critical sections of code. User processes, however, are time shared. A user process may be suspended at any time to allow another user process to run. We added a set of mutual exclusion locks to prevent multi-threading of critical sections of code.

The Unix kernel runs in a single address space. Our user space implementation runs in many different user address spaces, depending on which context happens to be active at the time a module is scheduled. Modules are intended to get pointers to their message blocks from a procedural interface, but there is no inherent reason why a module can't save a pointer to some piece of storage in a static data structure, and access it later from a different context. We constrain the Streams data structures to appear at the same place in every user address space context.

Several kernel data structures are available for inspection by Streams modules, including the u. area and the process table. These data structures would not normally appear in a user context. If the kernel provided procedural interfaces to inspect these data structures, better data abstraction could have been provided. As it is, our implementation must generate these data structures every time a stream is opened or a module is pushed, since the system has no way of knowing whether the module will access the data structure or not.

Module Loading

The Domain system boot loader is capable of loading object modules into user global address space. Any code or data appearing in this part of the address space is accessible at the same address from any user process. The procedure text (code) is simply mapped in, read-only, and demand paged from the file containing the object module. Data sections are loaded into per-process private address space, so that they appear at the same address in every process, but represent a different, private piece of storage for each process. This mechanism is used to implement global libraries.

Streams modules are not bound to any particular user process context. They expect that their static data sections will map to the same piece of real storage regardless of what context they are running in. We have modified the boot loader to load the static data sections of Streams modules into global read-write storage. The loader runs in user space and obtains its storage from a memory allocator that can allocate from either private or global space, so this was an easy modification to make.

Streams modules and drivers are added to a system by placing the object code that implements them into the `/sys/modules` directory. At the time the node is booted, the boot code loads any object modules it finds in this directory into user global address space. For convenience, sets of related modules and drivers are often bound together into a single object module.

Since the modules are loaded into user space and run in a user context, the standard debugger can be used to operate on the modules and their data structures. Although debugging in this way is not as easy as debugging a standard user program, it is not as difficult as debugging a kernel based Streams implementation would be.

Module Initialization

Each module and driver (or group of related modules and drivers) has an initialization routine that the boot program calls after loading the module. As part of its initialization, each module makes a call to declare itself to the rest of the system. One of the arguments to this call is a pointer to the `streamtab` data structure for that module. The `streamtab` contains pointers to information completely describing the module, including pointers to its `put`, `service`, `open` and `close` routines, and a pointer to the `module_info` structure. As each module is declared, the system builds a list of available modules in the `fmodsw` table, and keeps a list of pointers to the `streamtab` structures of those modules.

Each device driver also makes a call at initialization time to declare information about itself. This information is described in the section on device drivers.

The Domain I/O System

The Domain I/O system, or *IOS* [5], is based on an object oriented, UID named, typed file system. *Objects* (files and devices) are named by a Universal Identifier (*UID*), which can be

thought of as a device/inode pair that is unique across all machines for all time. Every object has a *type*, which is also named by a UID. When a file is opened for I/O, the open system call looks up the UID of the object and locates a *type manager* for that type of object. The type manager has entry points for each of the standard I/O *operations*, such as read and write. When a user program makes one of the standard I/O calls on an open stream, the system invokes the manager to perform the operation on that object. New types of objects can be defined and added to the system dynamically by writing a manager for that type and installing it in a well-known place (/sys/mgrs) without having to rebuild the kernel or reboot the machine.

Operations are grouped into logical sets, called *traits*, according to the generic kind of object they operate on. Open is part of the *OC* (open/create) trait. Read, write, seek, and the other generic I/O operations are part of the *IO* trait. All useful managers implement at least the OC and IO traits. Managers may also optionally implement any of several other pre-defined *auxiliary traits*. For example, all tty-like types implement the *TTY* trait, and all network protocol-like types implement the *socket* trait. The socket trait is modelled after the Berkeley socket call interface. As part of its initialization, each type manager declares to the system which traits it implements on the types that it supports.

The Streams Type Manager

Our Streams system provides a type manager that implements many of the pre-defined traits on behalf of all of the Streams device drivers. Each Streams device driver has an associated type UID which is stored in the file system. The file system implements a mapping between device names and the associated type UIDs. The Streams type manager currently supports the OC, IO, TTY, and socket traits for all the Streams device types.

As part of its initialization, the Streams type manager declares support for the OC trait for all Streams devices known to the system, as listed in the *cdevsw* table. The *cdevsw* table contains entries for all the streams devices configured for the node, and is built at boot time as the device drivers are loaded.

For each Streams device the type manager declares the auxiliary traits for that type of device as listed in the *cdevsw* table. Each module may also support one or more auxiliary traits. These traits are listed for each module in the *fmodsw* table, which is built dynamically as the modules are loaded. Support for these auxiliary traits is declared when a module is pushed on a stream. The list of traits supported by a particular device is kept on a per-process basis. If a device has had the TTY trait declared for it as a result of a TTY processing module having been pushed on top of it, for example, then only the process that pushed the TTY module will be able to use TTY operations on that device.

When an application opens a Streams device, the type manager's open operation does the work common to all the streams devices, including allocating a queue pair for the head of the stream, and then calls the device driver's open routine through the pointer in the *cdevsw* table.

The Streams type manager implements most trait calls by converting them to messages, sending them downstream, and awaiting a response. Some trait operations, such as creating a socket, require pushing a special module that interprets the messages peculiar to that trait. There is a duality between the procedure call oriented Domain trait mechanism and the passing of messages in the streams environment.

The operations of the transport library interface (TLI) have been recast as a trait to extend the functionality to the rest of the Domain system. It should be possible for those managers that now support the socket trait to also support the TLI trait by adding TLI operations to the manager.

All `ioctl` calls are mapped into the appropriate trait calls by the C library. If the stream is managed by the Streams type manager then the manager converts them to the proper `IOCTL` messages.

Device Drivers

Non-streams device drivers in the Domain system are implemented in user space, and are not necessarily connected to the I/O system. Their open, close, read and write entry points may be called by any user program directly, and typically present a device-dependent interface. To connect the device driver to the I/O system in a device-independent way, a type manager must be written. For non-streams devices, a separate type manager is usually provided for each type of device.

The interface between the type manager and the device driver is similar to the device driver interface in System V. The driver's initialization routine makes a device declaration call to export to the system the entry points for the `cdevsw` structure, which is built dynamically as drivers are loaded. The driver also exports an *event count* and a list of traits it is willing to support.

The event count is the basic process synchronization primitive in the Domain system [6]. System calls are provided to read, advance, and wait on event counts. Streams device drivers need not be real hardware drivers, but may be layered on top of some other system service, such as a low-level datagram service. These services typically export an event count to their clients. The driver itself has no process context associated with it, so it exports the event count to the Streams system, which will then call the driver's interrupt routine when the event count is advanced.

The Domain system currently has a limit of 32 event counts on which any process can wait at one time. This puts a global upper limit on the number of device drivers that may be open at one time, since all driver event counts are waited on by a single process.

Service Procedure Scheduling

The module scheduler is called by the type manager just before the return from any operation. At this time any module with work to do will get a chance to run. In addition, a server process provides a context in which timeouts and upstream events can run.

Each Streams driver exports an event count to the server. When an upstream event occurs, the event count is advanced, and the server wakes up and calls the driver's interrupt routine. Processing for downstreams events, generated as a result of some user program making a system call, run in the context of the process generating the event. This reduces context switch overhead, and can even eliminate the need to switch from user to kernel context in some cases.

Streams modules are subject to user process scheduling and suffer from synchronization problems. To provide the necessary synchronization, Streams queues and data structures are locked with a set of mutual exclusion routines provided by the underlying operating system [7]. The locks are transparent to the modules themselves. The current implementation locks all queues with a single lock.

Streams module procedures may execute in the context of the client process, the Streams server process or an unrelated user process. Downstream puts typically run in the context of the putting process. Upstream puts typically run in the context of the server process. A put called from a service procedure runs in the context in which the service procedure is being executed.

If a putting process enables a queue then the service procedures are usually run before returning from the system call. The execution of the service procedures may be delayed if another process had started the execution of the service procedures and was interrupted. The service procedures will then execute when the interrupted process regains control. At any particular time only one process can be executing service procedures and these procedures are executed in a round robin fashion.

The server process is started at boot time, after the modules and drivers are loaded. It is an ordinary user process. After performing some initialization work, it sleeps on a set of event counts, including all the device driver event counts and a timer event count. When it wakes up, it calls all the driver interrupt routines whose event counts have been advanced, runs any timeout routines that are due, and runs the module scheduler. The scheduler then runs any service procedures that have been enabled.

Configuration Parameters

The Domain implementation of Streams takes advantage of the virtual memory facilities to reduce the configuration work required of the system administrator. There are three classes of tunable parameters related to Streams. The approach taken for each class is given below.

Parameters that control the sizes of static tables, such as the maximum numbers of streams and queues, are made relatively large since these tables are allocated in virtual address space rather than wired memory as is the case in a wired kernel implementation.

Parameters that control the number of message buffers are assigned reasonable initial values but are extendible at run time with minimum run time penalty. Depending on the virtual memory size supported by the machine, there are upper limits placed on the buffer usage to preserve the flow control of protocol stacks.

Some parameters, such as the limits on the percentage of data blocks available at each priority level, are common to different protocol stacks running on the same node. The approach taken was to make them constant and known to protocol implementors so that the burden is placed on the implementors to make their protocols work with optimum performance. This is easier than the Unix System V model of making the system administrator choose values for these parameters.

Buffer Space Allocation

The buffer space for messages is allocated from user global space during the initialization of the type manager. The pool of buffers is automatically extended at run time if needed. The message blocks are not necessarily page aligned, and are pageable (not wired into physical memory). Device drivers that need messages in wired memory must do a final copy from stream allocated data blocks to private wired memory.

Older Apollo nodes have a 16 Mbyte virtual address space, of which 3 Mbytes is allocated to user global space. Most of this space is already occupied by the global libraries. On these machines the upper limit on total buffer space is limited to approximately 200 kbytes. Newer nodes all have at least a 64 Mbyte address space with plenty of unused global space.

Kernel Services and Data Structures

One of our primary goals was to be able to port Streams modules from Unix System V to the Domain system with a minimum of changes. Streams modules are written to run in a kernel

context, and expect to be able to use kernel services and data structures. Our user space implementation must then provide at least some of the kernel services these modules expect.

Streams modules in a kernel implementation run in an interrupt context. Since they can not make any assumptions about the context in which they run, they are unable to make any use of most of the data structures in the kernel. Driver and multiplexor open routines, however, do expect to run in the context of the user process doing the open. These routines may want to access kernel data structures that are not normally accessible from user space. We provide user space copies of parts of the u. area and process table for use by driver open routines. These copies contain only those fields that driver open routines might reasonably want to access. The structures are filled in by the Streams open procedure before the driver's open procedure is called.

Sleep() and wakeup() are implemented in terms of event counts. Sleep() can not be called except from driver open and close context, and attempts to do so are flagged as an error. Event counts are a generalized form of sleep and wakeup. An event count has a monotonically increasing value that can be advanced, or incremented, by a specified amount. A user process can read the current value of the event count, and can arrange to sleep until the event count reaches a given value. With the sleep and wakeup mechanisms, there is a timing window between testing some condition and sleeping until the condition is satisfied. This window is usually closed by writing a loop to sleep until the condition is satisfied. This results in the condition being tested at least twice. With event counts, the loop can be eliminated by reading the event count, testing the condition, then sleeping until the event count reaches a value greater than the value read.

Timeout() is implemented by the module scheduler. Timeout routines are scheduled to run in the same way that module service procedures are scheduled. The Domain user space timer event count only has a resolution of about 1/4 second, as opposed to the 1/60 second clock ticks that are standard in kernel implementations.

Malloc() and mfree() are implemented in terms of the standard user space storage allocator, using the global storage pool.

Evaluation

We believe that well-written modules should be easy to port to our Streams implementation. Well-written modules would only use those interfaces documented for use by Streams module implementors. Past experience suggests that real life programmers may not strictly adhere to these interfaces. The temptation, when writing code destined for a known implementation such as the System V kernel, is to make use of undocumented features. Programmers should resist this temptation when possible.

Porting device drivers is considerably harder, because of the inherent hardware dependencies involved. We have tried to make the interface from the I/O switch to the driver as standard as possible. Drivers that do not actually talk to hardware, such as pseudo-drivers and multiplexors, should be fairly easy to port.

The module and driver initialization routines are extra pieces that programmers must provide when porting to our Streams implementation. We have traded convenience to the programmer for convenience to the system administrator. Although the programmer must do some extra work to make the modules and drivers self-describing, the administrator only has to copy the object modules into a directory and reboot. We feel that this is a fair tradeoff.

Adding new modules currently requires rebooting, as they must be loaded into global space by the boot loader. There is no fundamental reason why the run time loader can't be made to load these modules into global space after boot time. This would make it possible to add new modules to a running system.

The current method for debugging global routines, such as Streams modules, is awkward. The code is normally mapped read-only, which makes it impossible to set breakpoints. It can be loaded read-write by giving a special flag to the boot program, but this introduces new problems. The breakpoint may be encountered in any process context, and unless it happens to be the context that is being debugged, the node crashes. Unfortunately, modules do not run in any predictable context. One possible fix would be to provide a debugging mode, in which all module processing would occur in the context of the server process.

The user space timer granularity of 1/4 second may be too coarse for some purposes. This does not seem to be a problem for our TCP/IP implementation, which uses the same timer mechanism.

There is currently work underway to implement a full communications protocol using Domain Streams. When this work is done, we will have a better idea of how good the performance is. We think that performance will be at least as good as that of existing protocols on the Domain system. Transmits potentially require no process context switching, as the downstream processing can occur in the context of the process that initiates the write. Upstream processing in some cases can be done in the context of whatever user process happens to be active, since the modules make no assumptions about which context they run in. Measurements on our TCP/IP implementation suggest that the reduction in context switch overhead may produce significant performance improvements. Further work is required to tune the system for optimum performance.

Acknowledgements

In addition to the authors, Larry Allen, Robert Israel, Tom LeMaire, Nat Mishkin and Eric Shienbrood participated in the design phase of this project.

References

- [1] Dennis Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, Vol. 63 no. 8 (October 1984).
- [2] David Olander, Gilbert McGrath, Robert Israel, "A Framework for Networking in System V," *Usenix Conference Proceedings*, Atlanta, Ga. (June 1986).
- [3] Andrew Rifkin, et. al., "RFS Architectural Overview," *Usenix Conference Proceedings*, Atlanta, Ga. (June 1986).
- [4] Jim Rees, Paul H. Levine, Nathaniel Mishkin, Paul J. Leach, "An Extensible I/O System," *Usenix Conference Proceedings*, Atlanta, Ga. (June 1986).
- [5] David P. Reed and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers," *Communications of the ACM* (February 1979).
- [6] *Using the Open System Toolkit to Extend the Streams Facility*, Apollo Computer Inc. (April 1986).

- [7] Edsger Dijkstra, "The Structure of the 'THE' Multiprogramming System," *Communications of the ACM* (May 1968).

Protocol Engine Design

Greg Chesson
Silicon Graphics

ABSTRACT

Local area network technology based on the 100 Mbit FDDI (Fiber Distributed Data Interface) ring network will soon be available. Forthcoming 1 Gbit fiber and related technology will bring an order of magnitude improvement to FDDI. Existing protocol standards are not well-matched to 100 Mbit or faster operation. System designers are beginning to consider alternative strategies. This paper discusses some of the issues involved in high-performance networking and describes a design effort to implement a transport protocol in VLSI.

Background

The state of the art in networking is, in all kindness, problematical. The new OSI protocols have promised a giant step - most likely a step in place, or slightly to the rear. At least one insouciant observer has referred to OSI as the Obsolete Systems Interconnect. Even though there are better ways of doing things, commitments to TCP and OSI create an inertia for preserving the status quo. Such grumblings are the normal output of creative minds confronting entropy death in the real world. There is hope.

The trend in network hardware is toward higher capacity media as we progress in 1988 from 10 Mbit/s ethernets to 100 Mbit/s FDDI (Fiber Distributed Data Interface) systems and later 1 Gbit/s networks. It is doubtful that network software can keep pace with these improvements. Figure 1 shows what happens to the 10 Mbit/s bandwidth of ethernet. Only about 6.7 Mb/s is available to a host at the MAC (Media Access) layer, in this case an ethernet interface chip. An IP layer running on a 68020 reduces the available bandwidth further, as does the TCP layer, leaving 1.2 Mb/s for the application.

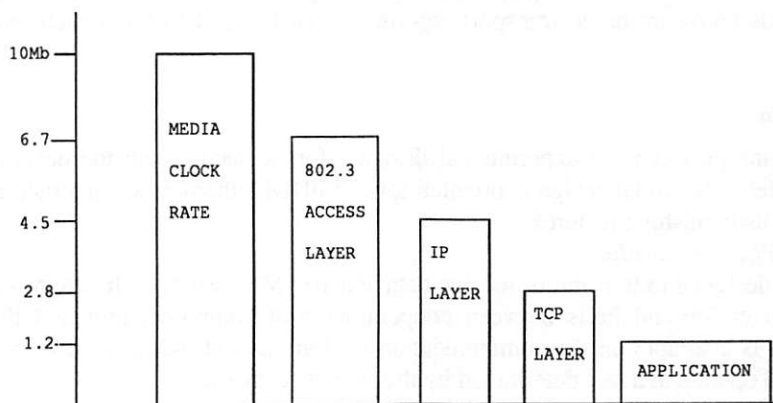


Figure 1

Suppose the media clock rate in Figure 1 is increased to 100 Mb/s. If the software, packet size, and protocol cpu are unchanged, the bandwidth available at an application program will *remain* essentially as shown in Figure 1. This is no great surprise since the software and system overheads are a limiting factor in the 10 Mb/s case. Incremental improvements can be made by using larger packets, improving software, and using cleaner protocols. However these improvements are not sufficient to "catch up" with

the network media.

Even though improvements in transport architecture are long overdue, it is the new network hardware that provides a catalyst for change. System designers now have a good reason for addressing transport problems. The concerns addressed by the current P-engine design start with performance and continue into operating system and internet areas as described next.

Numerous researchers have designed worthwhile transport algorithms. Examples include Lynch's *Alternating-Bit Protocol* in 1969, Sandy Fraser's *Universal Receiver Protocol* for the Datakit system at Bell Laboratories, and Dave Clark's *Netblit* design at MIT. Several important observations can be drawn from these and other designs. First, a protocol receiver - the part that processes incoming packets - must perform numerous checks and tests. Receivers are, almost by definition, more complicated than transmitters. System performance tends to depend strongly on receiver performance. Therefore protocol design should concentrate on simplifying the receiver. Second, timers are an unnecessary complication in a receiver. They can be eliminated in favor of placing any necessary timer functions in the transmitter. Lastly, a receiver can be further simplified by eliminating those parts of a receiver that would generate response messages to a transmitter. Instead, the receiver is made the "slave" of the transmitter, and generates response messages only when commanded to do so.

Distributed systems often prefer a datagram or message-oriented model of communication rather than the connection-based model provided by traditional transport protocols. It is also desirable for datagrams to be reliably delivered atomic messages. To suggest otherwise would require building another protocol above the basic datagram service. If a connection protocol is designed so that connections are created quickly - on the first message - then datagrams can be a special case of connections, i.e. short-lived connections. It is satisfying to use the flow and error control mechanisms of the traditional connection machine to provide reliable datagrams. Other operating system concerns are satisfied by providing a facility for transport users to view an end-to-end data stream as a sequence of typed messages, rather than a simple byte stream.

Current internet architectures have some interesting properties:

- 1) flow control protocols operate between communicating computers. The gateway programs, i.e. packet switches, between a pair of computers are not allowed to regulate data flows.
- 2) internet gateways are made of software and contribute to internetwork delays. Because of (1) they often become choke points.

These issues can be addressed by defining the flow and error control to be between gateways and between gateways and hosts. The result can be a low latency end-to-end network if the amount of buffer storage per gateway per connection is limited and if the gateways have a very low transit time. Hardware-based routing table algorithms plus hardware-based transport algorithms can be used to build such real-time gateways.

Protocol Engine Design

The protocol engine project is an experimental design effort to incorporate the design ideas mentioned above in a chip set. The initial design is oriented toward FDDI, ethernet, and internet applications. The design has several distinguishing features:

- 1) *12.5 Mbytes/sec bandwidth*

The PE is designed to transmit or receive data at a 100 Mbit/sec rate. It can move data at this rate on an end-to-end basis between cooperating host computers provided that sufficient bandwidth is available on the communication medium and at the host interface. Otherwise the PE will operate at a rate determined by the slowest component.

- 2) *VLSI Implementation*

Performance levels are achieved by implementing network, transport, buffer control, address and routing algorithms in a small number of integrated circuits.

- 3) *Multiple Physical Layers*

The P-engine is designed to operate with a variety of standard physical layers such as ethernet, 802.3, and the 100 Mbit/sec FDDI token ring.

- 4) *Real-time Gateway*

A pair of P-engines can operate in back-to-back fashion to provide real-time data transfers

between a pair of networks. This is useful for connecting dissimilar networks, such as ethernet and FDDI, as well as for expanding networks. The gateway architecture of the PE design avoids the internet congestion behaviors that arise in other systems by exercising flow control at gateways.

5) *Connections*

The P-engine implements a virtual circuit, i.e. connection-oriented, communications model. It controls the flow of data between computers and gateways, and corrects errors by retransmitting information. The architecture is designed to establish a virtual circuit context within the arrival time of a packet.

6) *Datagram Support*

Datagrams are represented as short-lived virtual circuits.

5) *Open Architecture*

PE technology will be made publicly available. This includes chips and software.

At first glance the emphasis on vlsi might appear to be a radical departure. However, networks will surely evolve in this direction, gradually replacing software by vlsi chips. This approach will eventually bring the same high degree of interoperability and performance now associated with Physical Layer hardware to network transport.

Even though the PE represents a large incremental step in networking, there is little risk involved. The design builds on the ideas of existing research networks and utilizes currently available semi-custom chip technology.

Design Strategy

The receive side of a network interface is critical to system performance. The sum of incoming traffic from multiple senders to a server or gateway can saturate a reader not designed to handle the load. Therefore the PE receiver is designed to accept the maximum FDDI bandwidth of 100 Mb/s or 12.5 MB/s.

The PE should be able to *sustain* a 12.5 MB/s rate between host and network. This means that PE data buffers must operate at 25 MB/s. Address and route processing, packet formation, message assembly, flow and error control operations must also be performed at this rate. That is, it must be possible to perform all buffering and transport operations associated with an incoming packet within the packet arrival time period. This time is the number of bytes in the packet times 80 nanoseconds. If the smallest packet is 128 bytes, then we have about 10 microseconds to complete packet processing.

Inability to complete processing within a packet time means that fifo buffers in the network interface may fill, leading to dropped data and retransmissions. This is a serious concern on an FDDI-class network because data can arrive at a rate which is an order of magnitude or more greater than existing networks. The P-engine strategy is strongly oriented toward processing all incoming packets in real-time.

The current P-engine design has two packet formats, shown in Figure 2.

CODE	ADDRESS	DATA	SEQ	LENGTH	TYPE
header			trailer		

Data Packet Format

CODE	ADDRESS	SEQ	ACK	WINDOW	MSG	ID	TIME
------	---------	-----	-----	--------	-----	----	------

Control Packet Format

Figure 2

The data packet consists of a header, data segment, and trailer. The header contains a packet command code and addressing information. Command codes tell the receiver to buffer user data, to generate response messages, and to establish or remove a connection/datagram context. The initial packet of a connection may contain additional address information in place of the user data field. This permits connections/datagrams to be passed through a P-engine node to a network with a different or more complex connection setup structure. The data segment is variable length. On FDDI the minimum data segment is 64 bytes to guarantee at least 5 microseconds processing time between the header and trailer. On ethernet or other slower networks, the minimum segment could be relaxed or a cost-reduced and slower P-engine would be used. The trailer contains a packet sequence byte, a length field for the data segment, and a user-controlled packet type field. The type field is intended for delimiting messages, providing out-of-band capability, and data typing for user processes.

The control packet format contains flow, error, window, message, and timestamp information important to the link protocol but not needed with every data packet. The flow of information between a pair of P-engines will usually consist of data packets in one direction and control packets in the opposite direction. Minimizing the non-user-data overhead in this way simplifies the protocol logic somewhat - there is less information to discard in each packet - and improves the protocol performance when P-engines are applied to slower networks or point-to-point links that have less bandwidth than FDDI.

Figure 3 illustrates the P-engine receiver strategy for dealing with incoming packets in real-time. A pair of back-to-back similar packets are shown along with P-engine internal operations in timing diagram form. The idea is to perform the address lookup while buffering incoming data. The address lookup should complete in time to process control information at the end of the packet.

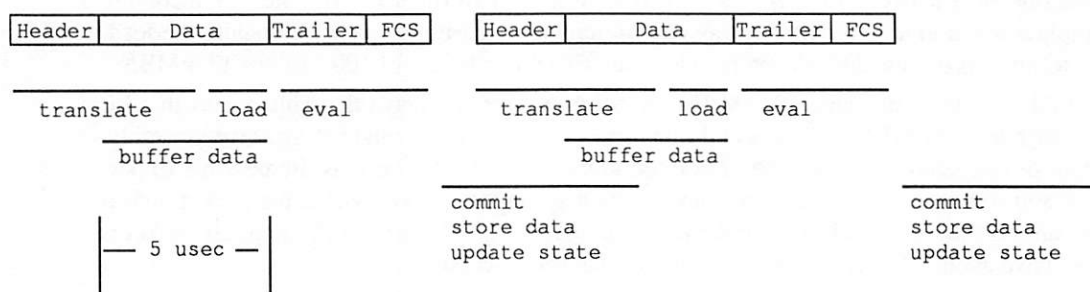


Figure 3

The frame check sequence (FCS) is defined by the underlying network hardware. It is shown here to emphasize that information in an arriving packet is not "dependable" until the FCS error check has been passed. Therefore no receiver or buffer state changes should be made until after the FCS check. The operations in Figure 3 are as follows:

translate

Each active connection has a state vector in P-engine memory. The address translation step uses the header address as a key to "lookup" the appropriate state vector. The results of the lookup may be (1) to return the state vector, (2) to recognize the address as a new connection and return a newly created state vector, (3) to reject the packet.

load

The result of the address translation is used to load the state vector into the P-engine.

eval

The incoming trailer is checked against packet length and expected sequence number.

buffer data

Incoming data is put into a holding area pending a decision to accept or reject the packet.

commit

After the FCS check the receiver will either discard the input packet, making no change in the local state except for counting errors, or accept the packet. If the packet is accepted the data is appended to the proper queue and local state is updated.

The P-engine design strategy starts with a receiver architecture that meets the design goals. The transmitter design then follows, almost as a deductive exercise, complementing the receiver concepts.

Transmitter tasks consist of building output packets and "operating" the remote receiver. Transmitter logic constructs the non-data portion of a packet directly from the state vector. There are subtleties in the packet design that are advantageous for both transmitter and receiver. The length field is used by the receiver as a check. The transmitter is designed to burst data from buffer storage to the network without having to know the final packet length. The data burst stops when a maximum is reached or when the buffer logic indicates end-of-message. The minimum size constraint on data segments lets the transmitter prepare the trailer information while the data is streaming out to the network. The transmitter logic then appends the sequence number, length field, and type. This model of interaction between transmitter and buffer logic simplifies the amount of real-time chitchat between the two components - an important consideration at 100 Mbit data rates.

The transmitter is responsible for observing round-trip delays, adjusting connection timers, and performing retransmission. Because the P-engine must manage a buffer queue for each active connection/datagram, the queue manager logic must handle a set of input and output queues. It is worth noting that the logic for managing the queues doesn't really care about their contents. Therefore the hardware mechanism for keeping track of the transmitter tasks - timers and work lists - can be based on the use of the data queue manager as a task list handler.

Chip Set Architecture

The P-engine idea was conceived first as a single chip architecture; that is, one large custom chip plus some memory that could easily be added to existing network controller designs. This remains an eventual goal; however, other goals conflict with the single chip plan. Gate arrays or semi-custom technology are more appropriate for quickly building a system than the full-custom approach. Given the complexity limitations of semi-custom technology compared to full-custom parts, the design had to be partitioned into a chip set. As it turned out the timing constraints of handling FDDI data require simultaneous operation of several data paths: data buffers, address translation, state vector, and host interface. A single chip would need an inordinate number of i/o pins to operate all these data paths at the same time: another reason for a multi-chip approach.

The bandwidth requirements at ethernet rates can probably be satisfied by a single data path. In this case the initial vision of a single chip implementation becomes possible. Given a successful multi-chip prototype, it is likely that a lower-end single chip design will be produced.

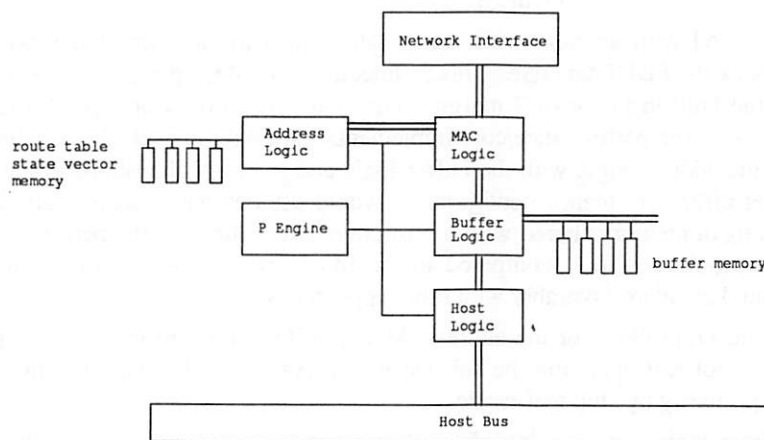


Figure 4

The current design for a P-engine is shown in Figure 4. Three components - P-engine, Address Logic, and Buffer Logic - are the core of the system. Mac logic is specific to a particular network and would require a new design for each situation. Host logic is specific to a particular host interface style, although it appears that a single chip design can cover the common and useful cases.

In this diagram the double lines represent the high-speed data path between network and host. The other lines connecting the P-engine to other components represent inter-chip data and control. Mac logic extracts incoming address fields and passes them to the address logic. It also routes the data segment to the buffer logic, and passes trailer data to the P-engine.

The address logic carries out a multi-tree, or trie, traversal based on one tree level per 4-bit address nibble. Routing table space required for this algorithm is 16 table words per nibble, i.e. 32 words per address byte. Total worst case space required to translate 4096 10-byte addresses would be 128K words - a reasonable amount for a large scale implementation. Table space for 128 addresses would be 4K words - a reasonable amount for a workstation-scale implementation. Time required to translate an address string would be about 80 nanoseconds per nibble, or about 1.6 microseconds for a 10-byte address. These times are consistent with the receiver timing constraints discussed earlier.

The buffer logic for FDDI shown in Figure 5 utilizes the parallel data paths available on video rams. Data to or from the network moves in or out of the shift register port on the video rams. Buffer logic can freely access the dynamic ram portion of the vrams while the shift register is operating. In a minimal design the vram array would have 1K bytes of shift register. While 1K of data is moving in or out of the shift register - about 81 microseconds at FDDI speeds - the buffer logic has sufficient time to serve host and P-engine memory accesses. At the same time the buffer logic sets up broadside transfers between internal shift registers and ram arrays.

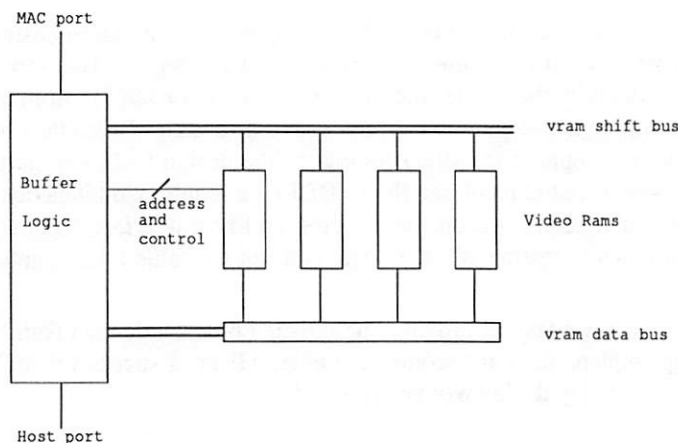


Figure 5

If the vram array is organized with an 8-bit wide serial data path, current vrams have twice the bandwidth needed to source or sink the FDDI data rate. This architecture should keep pace with faster successors to FDDI by means of the built-in factor of 2 margin, plus next generation vram speed improvements, plus a wider data path. A lower performance/cost implementation might conceivably continue to use vrams, but would combine the address logic with the buffer logic and run both algorithms in the same vram memories. An even lower cost/performance configuration would use combined address and buffer logic running the same algorithms in memory shared with a host computer. Although the performance of these alternative configurations would be "low" compared to the full-blown multi-chip arrangement, it appears that the performance could compare favorably with other approaches.

The P-engine itself contains controllers for the address, MAC, buffer, and host components plus a state machine for doing the protocol and operating the subsystem components. The state machine has a some programmability, limited primarily by chip real-estate.

Different buffer and address logic configuration ideas illustrate the idea of *scaling* the architecture for different environments. The potential *range* for the multi-chip design is fairly broad. Some configurations include:

- *no chips*

- work is underway on an all-software implementation of the P-engine protocol and algorithms,

- *subsystems*
address translation hardware could be added to a microprocessor-based system,
- *link set*
the chip set, minus address translation logic, could be used to operate a dedicated point-to-point link.
- *chip set 1*
the chip set could be implemented on a standard i/o bus, which might limit performance to a few Mbytes/sec,
- *chip set 2*
the chip set could be used on a high-speed bus interface of the kind found inside high-end workstations and super-mini systems,
- *chip set 3*
the chip set could be configured with enough memory to support several thousand active connections for a high-capacity network server or gateway.

Figure 6 shows how P-engines could be combined to build an internet gateway. An FDDI ring is shown connected to a pair of ethernet. Each P-engine would consist of a circuit board. The circuit boards would communicate over a system bus. With P-engine host logic designed to operate in a multi-host environment, multiple P-engines would appear to each other as multiple separate hosts sharing the bus.

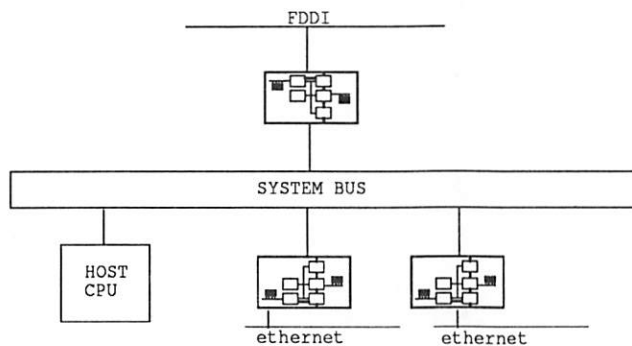


Figure 6

A host cpu would be needed on the system bus for supervisory functions, but would not be active in the data path between P-engines.

Conclusions

High-speed networks in the 100 Mbit/s range provide an opportunity to examine current practices in network transport. The P-engine project shows how vlsi technology can address performance issues. This paper demonstrates a methodology, based on receiver design, to structure a transport protocol frame format for high-speed processing. The discussion also points out areas of neglect in current network transport design, in particular real-time gateways, fast connection handling, and reliable datagrams.

Acknowledgements

The Protocol Engine is as much a product of the author's personal environment as it is a result of the present technology environment. Particular credit should be given to Silicon Graphics for encouraging skunkworks and for understanding that some developments are more useful when they can be made non-proprietary. Sandy Fraser of Bell Laboratories deserves special acknowledgement for specific technical suggestions and support. Larry Green, co-chair of the ANSI XT39.5 FDDI committee, has championed the P-engine idea within the FDDI community. Many friends and colleagues have provided motivation for this work by suggesting that it seems crazy enough to actually work.

Virtual Address Cache in UNIX†

Ray Cheng

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, Ca. 94043
rcheng@Sun.COM or sun!rcheng

Most of the cache memories in computer systems are addressed by physical addresses. In systems that support virtual addresses, this means that the cache is accessible after a virtual to physical translation is done. In order to reduce the cache access time, Sun-3 Series 200 workstations include a cache accessed by virtual addresses. However, unlike physical address caches, virtual address caches are not transparent to software. Data consistency problems arise when there is a change to the virtual to physical mapping or when two or more virtual addresses map to the same physical address.

To hide this data consistency problem from application programs, the kernel should ensure that a program runs on a machine with a virtual address cache produces the same result as the one it produces when it runs on a machine without a virtual address cache. To guarantee such system correctness, the kernel maintains the following invariant: if an entry is in the cache, its virtual to physical mapping must be correct and no other virtual address maps to that physical address without going through the same cache entry.

There are three things the kernel can do to satisfy this condition. The first is to flush an entry from the cache when its virtual to physical mapping becomes incorrect, e.g. when the mapping of a page of the out-going process becomes incorrect during a context switch. The second is to make virtual addresses differ by modulo 128K when we set up a virtual address that maps to the same physical address as another virtual address, e.g. in the implementation of AT&T System V shared memory. Finally, if the kernel doesn't know when to flush the cache and the assignment of virtual addresses that map to the same physical address is beyond the kernel's control, e.g. in the implementation of the mmap routines in 4.2BSD UNIX, the kernel makes these virtual addresses non-cacheable.

This paper first gives some background in cache memories and the Sun-3 cache architecture in particular. It then describes the approach we took and the new routines we added to the kernel. Several examples illustrate why and how some entries are flushed from the cache when a mapping is invalidated, when a mapping becomes invalid implicitly, and when the protection attributes of a mapping are changed. Since flushing an entry from the cache takes up to several hundred microseconds, we avoid cache flushing as much as possible. This paper describes several cases where cache flushings are avoided even when the mappings become incorrect. After that, the uses of modulo 128K addressing and non-cache pages are discussed. The debugging turned out to be as difficult as we had feared, especially due to the fact that we debugged the kernel and the cache hardware at the same time. Finally, the overhead of cache flushings as well as their measurement are discussed.

† UNIX is a registered trademark of AT&T

1. Introduction

To increase effective memory access speed, modern computer systems use a cache memory in front of main memory. Almost all computer systems address cache memories by physical addresses. This means that a Translation Lookaside Buffer (TLB) lookup and possibly a virtual-to-physical translation is needed before the information in the cache memory can be accessed.

In Sun-3 200 series machines, the cache memory is addressed by virtual addresses. The advantage is that TLB is eliminated and the cache can be accessed without a virtual-to-physical mapping. However, this cache is not transparent to software. To relieve users of the need to modify their programs, the operating system has to make the cache transparent to user level programs. In this paper we present our experience with the extension of a 4.2BSD based UNIX kernel for a machine with such a cache memory.

2. Background

Cache memories are small, high speed memories used to hold information that is believed to be currently in use [Smith, 82]. Information located in cache memories can be accessed in much less time than that located in the main memory. Thus, a CPU with a cache memory spends less time waiting for instructions or data to be fetched or stored.

When a write operation is performed, main memory can be updated in two fashions. In the first method, the cache memory receives the write and the main memory is not updated until that cache line is replaced. This method is known as copy-back (or write-back). The second method, known as write-through, updates both the cache memory and the main memory when the write is performed.

Cache memories can be addressed either by physical addresses or by virtual addresses. The cache memory that is addressed by physical addresses is called a physical address cache while the cache memory that is addressed by virtual addresses is called a virtual address cache. Their conceptual difference relative to the virtual-to-physical address translation is illustrated in Figure 1.

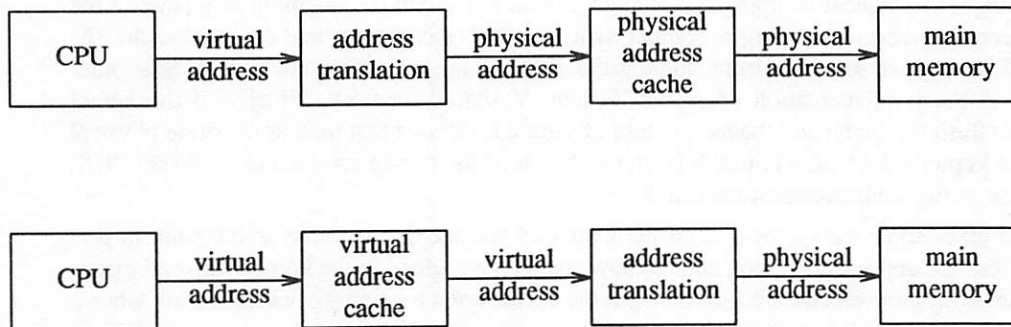


Figure 1 physical address cache vs. virtual address cache

3. Sun-3 Cache Architecture

The Sun-3 virtual memory architecture provides each process with a 256 megabytes virtual address space [Sun, 86]. The Sun-3 MMU uses a page size of 8K bytes and segment size of 128K bytes. The MMU consists of eight distinct address spaces or "contexts", each of them has a size of 256M bytes. The kernel is responsible for multiplexing the hardware resource of contexts among the set of processes it supports, using each context to represent a process's virtual address space.

The Sun-3 cache is a virtual address, write-back cache. The cache is 64K bytes in size with 16 byte lines. It has a context field in its cache tags to distinguish the eight contexts of the Sun-3 MMU. Hence the entire cache is not wiped out on a context switch. When a virtual-to-physical mapping is set up in the MMU, a page may be made non-cacheable. If a page is made non-cacheable in the MMU, the information

in this page will not be put in the cache.

To make it possible for the system to work correctly with the virtual address cache, the architecture includes three cache flush operations: the page match flush, the segment match flush, and the context match flush. These flush operations flush all cache lines whose tags match a page address, a segment address, or a context number. When a cache line is flushed, if that line is modified and valid, a write-back to the main memory is done. In addition, if the line is valid, it becomes invalid after being flushed. (A hit occurs when the requested virtual address matches with the virtual address of a *valid* line as well as the type of the access satisfies the protection check.)

Further, the Sun-3 cache guarantees that all virtual addresses that map to the same physical address are put to a common cache location if their (virtual) addresses differ by a multiple of 128K bytes. This applies to virtual addresses within the same context or between different contexts.

4. Problems with Virtual Address Caches

The correctness criterion for introducing a cache is that any program run on a system with a cache should produce the same result as it produces from a system without such a cache. A system with a virtual address cache introduces two kinds of data consistency problems: *mapping change* and *synonyms*.

4.1. Mapping Changes

Mapping change introduces data inconsistency as follows (Figure 2): At time t_1 , virtual address v maps to physical address p_1 and there is a write operation that writes value x to the content of virtual address v . This value x is associated with virtual address v in the cache. At a later time t_2 , the mapping of v is changed such that it maps to physical address p_2 which contains a value y . Then, the CPU issues a read operation to v . On systems without such a cache, the value y should be the result of this read access. However, if $\langle v, x \rangle$ is valid in the cache, the value x will be the result of the read access. Furthermore, the value x is not written to the physical memory p_1 .

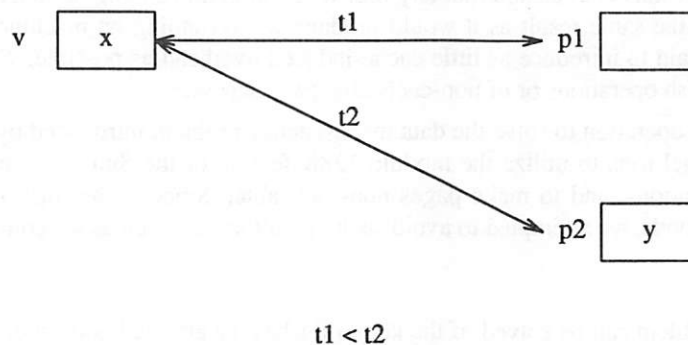


Figure 2 Data Inconsistency due to Mapping Changes

4.2. Synonyms

Synonym is the case when there is more than one virtual address mapped to the same physical address. Synonyms introduce another kind of data inconsistency as follows: Virtual addresses $v1$ and $v2$ both map to physical address p . At time $t1$, the CPU writes the value x to $v1$. At time $t2 > t1$, the CPU writes the value y to $v2$. (Figure 3) At this time the value x is associated with $v1$ in the cache while the value y is associated with $v2$ in the cache. The physical memory p should contain the value y since it is written at the later time. Next, at time $t3 > t2$ the CPU reads from $v1$. Since $\langle v1, x \rangle$ is stored in the cache, the value x is returned to the CPU. However, on systems without a virtual address cache, the value y is returned to the CPU.

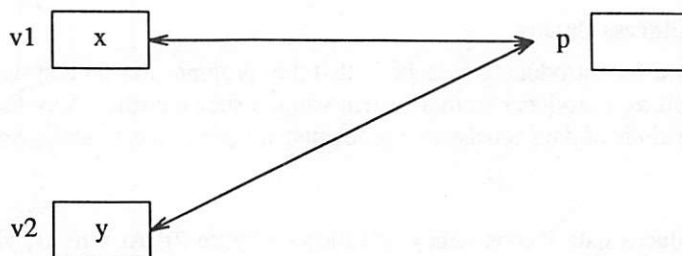


Figure 3 Data Inconsistency due to Synonyms

5. Unix Kernel Extensions

The goal of the kernel extensions is to ensure that any user level program running on machines with a virtual address cache produces the same result as it would produce when running on machines without such a cache. Also, attention is paid to introduce as little cache-induced overhead as possible. Such overhead can be the result of cache flush operations or of non-cacheable page accesses.

The kernel uses cache flush operation to solve the data inconsistency problem introduced by mapping changes. For synonyms, the kernel tries to utilize the modulo 128K feature of the Sun-3 cache architecture, to execute cache flush operations, and to make pages non-cacheable. Since cache flush operations take tens to hundreds of microseconds, we attempted to avoid such operations as much as we could.

5.1. Mapping Changes

This data inconsistency problem can be solved if the kernel flushes the affected portion of the cache whenever there is a mapping change. As shown in Figure 4, before the mapping v -to- $p1$ is changed to v -to- $p2$, we issue a cache flush operation to virtual address v . Then, the value x is written physical memory $p1$ and there is no valid entry for virtual address v in the cache. When the CPU reads from virtual address v after the mapping v -to- $p2$ is set up, since v doesn't have a valid entry in the cache, the read causes a cache miss and the value y is obtained from the physical memory.

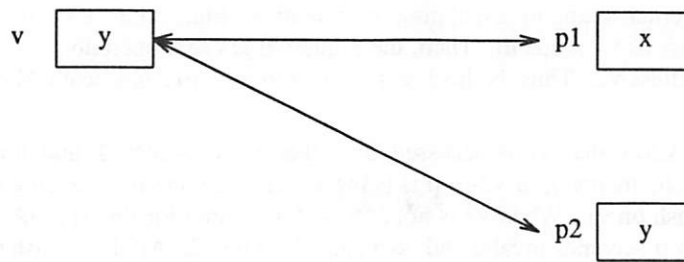


Figure 4 Mapping Changes with cache flushes

If the mapping of v-to-p1 is invalidated before the mapping v-to-p2 is set up, as occurs in lots of places in 4.2BSD Unix, we only have to flush the cache when a mapping becomes invalid. The mapping change from invalid to v-to-p2 doesn't need a cache flush, because there is no valid entry for virtual address v in the cache. We generalized this cache flushing strategy as follows: There is always a cache flush operation when a mapping is changed from valid to invalid, but there is no cache flush operation when a mapping is changed from invalid to valid. This saves a number of unnecessary cache flush operations while maintaining the correctness criteria.

Example 1. In SunOS Release 3.2, the u page is in the kernel virtual address space. This kernel virtual address maps to the physical u page of the running process. Therefore, when a process is scheduled to run during a context switch, a mapping from virtual address `_u` to the physical address of its u page has to be set up. Similarly, such a mapping is invalidated when a process is "switched" out during context switch. To avoid the data inconsistency due to this mapping change during context switch, we do a page match flush when the mapping of u page is invalidated. (We don't do a page match flush when a new mapping for the u page is being set up.)

Example 2. In `pageout()`, the pageout daemon marks not-recently-used pages to be invalid. If there is no page match flush for this page and the MMU mapping is invalidated accordingly, a subsequent write-back of this page to the physical memory will fail.

There are also cases where a mapping is not released or invalidated explicitly. To follow our cache flushing discipline, namely, flush when a mapping becomes invalid not when a mapping becomes valid, we deem the mapping to be invalid when this mapping is not used any more and start the flush operation at this time. For example, `forkutl` is used to map to the physical u page of the child process when the kernel is servicing the `fork(2)` system call. This mapping is not released explicitly when the routine using `forkutl` returns. However, we deem the mapping to be invalid before the routine returns. Therefore, the page match flush for `forkutl` is done before the routine returns.

5.2. Synonyms

If the virtual addresses in a case of synonym can be set by the kernel, such virtual addresses are set such that their differences are modulo 128K. Thus, the Sun-3 cache architecture guarantees that all such virtual addresses that map to the same physical address occupy the same cache line in the cache memory. This solves the data inconsistency problem without any cache flush overhead. This technique was used in the implementation of the System V shared memory in SunOS Release 3.2.

Many times it is inconvenient to assign values to virtual addresses. For example, it is impractical to allocate kernel global variables that may be used to map to same physical addresses at addresses that differ by a multiple of 128K. However, if the kernel knows about the access pattern of synonyms `v1` and `v2`, we can treat the other mapping as invalid while one mapping is actively in use. Then we can flush the virtual address when its mapping becomes invalid. The following example illustrates this method.

Example 3. In SunOS Release 3.2, a Direct Virtual Memory Access (DVMATM) operation from virtual address v1 starts with the kernel setting up a mapping from another virtual address v2 in the DVMA region to the same physical address as v1 maps to. Then, the requested DVMA operation, either read or write, is started through virtual address v2. Thus, both v1 and v2 map to the same physical address, a case of synonyms.

However, in this case, we know that v1 is accessed first, then v2 is accessed, and finally v1 is accessed again (Figure 5). When the mapping of v2-to-p is being set up, we view the mapping of v1-to-p as invalid and do a page match flush on v1. When v2 is not accessed any more for this DVMA operation, we view that the mapping of v2-to-p becomes invalid and do a page flush on v2. All these flush operations turned out to be necessary both on DVMA read operations and on DVMA write operations.

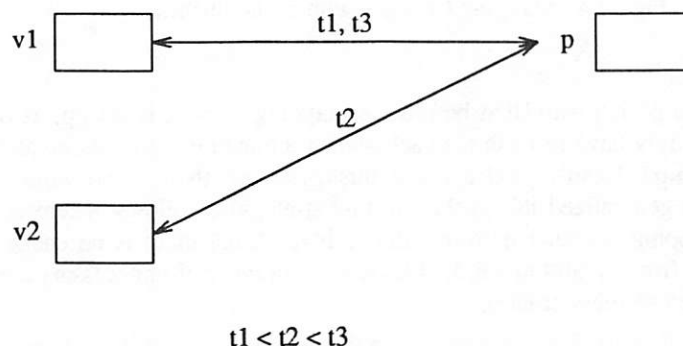


Figure 5 Synonyms with cache flushes

5.3. Don't Cache Page

If in a case of synonyms, we can neither set the virtual addresses nor know about the access pattern of the virtual addresses, we make all these virtual pages non-cacheable. As discussed in Section 3, any page that is made non-cacheable in the MMU is not included in the cache. On Sun-3 200 series implementation, when a page is non-cacheable, its access is much slower than the the speed CPU can access memory. Thus, a number of wait states are needed. Therefore, this method is used as the last resort to guarantee system correctness.

Example 4. The mmap(2) routine from 4.2BSD Unix allows different user level programs to map to the same physical address. In this case, user level addresses are determined by user programs and their access behavior is unknown to the kernel. As a result, the kernel makes these user pages non-cacheable. If the kernel page is also used by a device driver, the device driver should make the shared kernel page non-cacheable also (Figure 6).

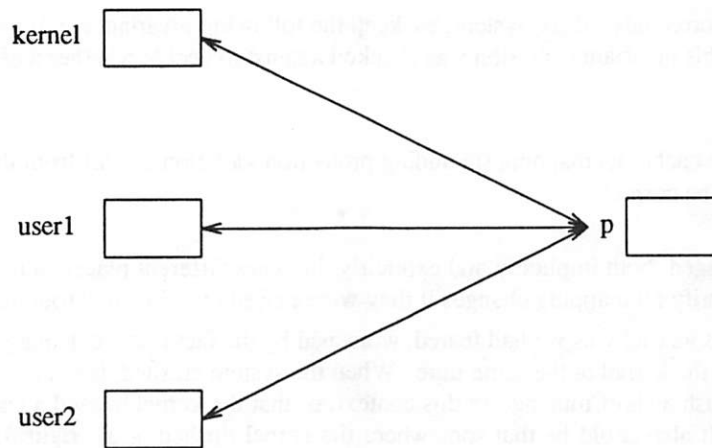


Figure 6 Synonyms that Require the Use of Don't Cache Pages

5.4. Don't Flush if Not Necessary

The cache flush operations are rather time consuming in our implementation. Therefore, we avoid flushing as much as possible. One trick in the case of u page flush is that since only half of an 8K bytes pages is really used, the kernel flushes 4K bytes instead of flushing the entire page. There are a number of cases where flush operations can be avoided when a mapping changes from valid to invalid. For example in `swap()` of 4.2BSD Unix, dirty pages are mapped to the context of `proc[2]` and hence invalidate the previous mapping. Since dirty pages have been flushed in `pageout()` already, there is no need to flush these pages again in `swap()`.

6. Performance

In order to measure the cache-induced overhead, we added instrumentation code to the kernel to record the total number of each kind of flush. Then we wrote a user level program that reads these kernel numbers. Next, before and after we ran benchmark programs we probed the number of each type of flushes. The differences of these numbers approximate the number of each kind of flush occurred from running this benchmark program.

Though hardware engineers are able to give us the minimum time needed to do each kind of flushes, the time that a flush really takes depends on the number of lines being modified at the time of flush and on other system activities such as Ethernet traffic and VMEbusTM activities. Lacking analytical data, we decided to use a logical analyzer to measure the average time needed to do a flush. Since different benchmarks cause the cache lines modified quite differently, we measure the average flush time separately for each benchmark we ran. We also estimated the time spent in software to instruct the hardware to do a flush. Next we multiplied the number of each kind of flushes by the average time of each kind of flushes to get an approximate total time spent in flushing the cache for a benchmark program. Lastly, we divided the time spent in flushing the cache by the total amount of time spent in running the benchmark program.

We ran one benchmark program at a time on lightly loaded multi-user mode. For the dhrystone benchmark, only 0.13% of total time was spent in flushing the cache. Most of the flushes were to flush the u page during context switches. Also, the u page was barely modified hence the cache hardware spent minimal time doing write-back. Another benchmark program which causes page faults to occur continuously spent 3.0% of total time in flushing the cache. In this benchmark, paging to and from the disk using DVMA operations caused many page match flushes. Also, almost all of such flushes needed write-back operations.

7. Conclusions

To guarantee the correctness of the system, we keep the following invariant condition true at all time. Throughout the design, this invariant condition was checked against to decide whether a cache flushing was needed.

If an entry is in the cache, its mapping (including protection violation check) from the virtual address to physical address must be correct.

Mappings are changed, both implicitly and explicitly, in many different places in the 4.2BSD kernel. It would be easier to identify all mapping changes if they were placed only in a few routines.

The debugging was as tricky as we had feared, worsened by the fact that we debugged the kernel and the cache hardware with the kernel at the same time. When the system crashed, the cause could be that the kernel missed a cache flush a short time ago in this context, or that the kernel missed a cache flush several context switches back. It also could be that somewhere the kernel flushed at the right time but flushed a wrong address. Still, as it sometimes turned to be, it could also be that the cache hardware didn't flush the cache as it should.

Finally, it was nice to see the system ran faster than we had anticipated. Also, the flush overhead was found to be smaller than we expected.

Acknowledgements

Ed Hunter provided enormous help in the debugging stage, especially in using logical analyzer to identify hardware bugs to the hardware people. Jo Moran and Bill Shannon helped to find the last (known) bug.

8. References

[Smith, 82]

Smith, Alan Jay, "Cache Memories", *ACM Computing Surveys*, September 1982.

[Sun, 1986]

Sun Microsystems, Inc., "Sun-3 Architecture: A Sun Technical Report" August 1986

UNIX™ on a VLIW

*Patrick Clancy
Benjamin F. Cutler
J. Christopher Dodd
Douglas W. Gilmore
Robert P. Nix
John J. O'Donnell
Christopher P. Ryland*

Multiflow Computer, Inc.
175 North Main Street
Branford, CT 06405

1. Introduction

Multiflow Computer, Inc. was founded in 1984 to develop high-performance general purpose computer systems based on two fundamentally new technologies: Trace Scheduling™ compacting compilers and Very Long Instruction Word (VLIW) architecture.

Multiflow's TRACE processors include up to 28 functional units operating simultaneously, in a single execution stream, under the control of a Very Long Instruction Word. Trace Scheduling compacting C and FORTRAN compilers exploit fine-grained parallelism throughout programs, rearranging operations to execute concurrently.

Because VLIW architecture expresses parallelism at the level of individual computational steps, TRACE systems deliver high performance regardless of the structure of the application; the speedup from parallelism is transparent.

The operating system developed for the TRACE is TRACE/UNIX, an enhanced version of BSD 4.3. The complete set of UNIX utilities has been ported to the TRACE, and gets the full performance benefit of Multiflow's compiler and hardware technology. The TRACE/UNIX kernel has undergone various modifications and refinements at Multiflow, first to get it running on the unique TRACE hardware, and then to enhance its performance and functionality to meet the expected requirements of the supercomputer user community.

We will discuss the TRACE architecture in section 2, and the TRACE/UNIX kernel in section 3. Particular attention will be paid to the way the kernel deals with unique aspects of the TRACE architecture, the I/O subsystem, and kernel enhancements.

2. The Trace Architecture

The core ideas which led to Multiflow's technology were developed by Joseph A. Fisher while a graduate student at NYU in the late '70s [2], and were developed and demonstrated in the ELI project conducted at Yale from 1979 to 1984 [3, 4]. VLIW architecture was developed hand in hand with Trace Scheduling compacting compiler technology.

VLIW architecture incorporates many pipelined functional units and datapaths. All units run in lockstep; the machine is completely synchronous. Each functional unit is controlled by a dedicated field of the Very Long Instruction Word. There are no bus arbiters, queues, or other hardware synchronization mechanisms.

Trace Scheduling, TRACE, and Multiflow are trademarks of Multiflow Computer. UNIX is a trademark of AT&T Technologies. VAX and VMS are trademarks of DEC.

A single program counter and flow of control directs the fetching of instructions. Each functional unit initiates one operation specified by its instruction word field during each clock cycle.

The full time cost of each operation is exposed at the instruction-set level. Each operation type takes a fixed time to complete. All functional units are fully pipelined, allowing new operations to start on every unit in every instruction.

The architecture is *load-store*. Memory references are explicit; all other computations use general registers to hold operands and results.

There is no microcode. Hardware directly executes the Very Long Instruction Words, without the overhead of intermediate interpretation or decoding steps.

From a hardware standpoint, VLIWs may be viewed as generalized vector machines. The pipelined functional units are essentially the same as might be found in a vector machine; but no separation between scalar and vector hardware exists. Hardware control units which count out vector addresses have been replaced by wide instruction words, which specify each computation uniquely. Logic has been replaced with memory.

Alternatively, VLIWs may be viewed as RISC machines with overlapped execution gone wild. The instruction words allow the expression of arbitrary execution overlap among scalar operations, with potentially very large numbers of operations executing simultaneously.

2.1. Overlapped Execution

Computer designers have attempted to apply parallel hardware to speed execution since the earliest computers were designed. Substantial cost and reliability improvements are obtained when high-volume, low-cost electronics can substitute for smaller numbers of exotic, high-power circuits.

Most modern computers use small scale parallelism to great advantage; for example, instruction *fetch* is commonly overlapped with instruction *execution* in modern "pipelined" superminicomputers. Instruction fetch pipelining, coupled with appropriate instruction set design, helps computers achieve a major design goal: executing one instruction per clock cycle. As RISC design philosophy spreads throughout the industry, this goal is increasingly achieved by midrange computer systems.

Designers of high-performance computers have achieved a more difficult objective: executing multiple instructions per clock cycle. Modern high-performance mainframe computers incorporate a technique introduced in the late 1960's: *overlapped execution* of multiple program steps. Overlapped execution exploits parallelism among individual scalar operations, such as adds, multiplies, and loads.

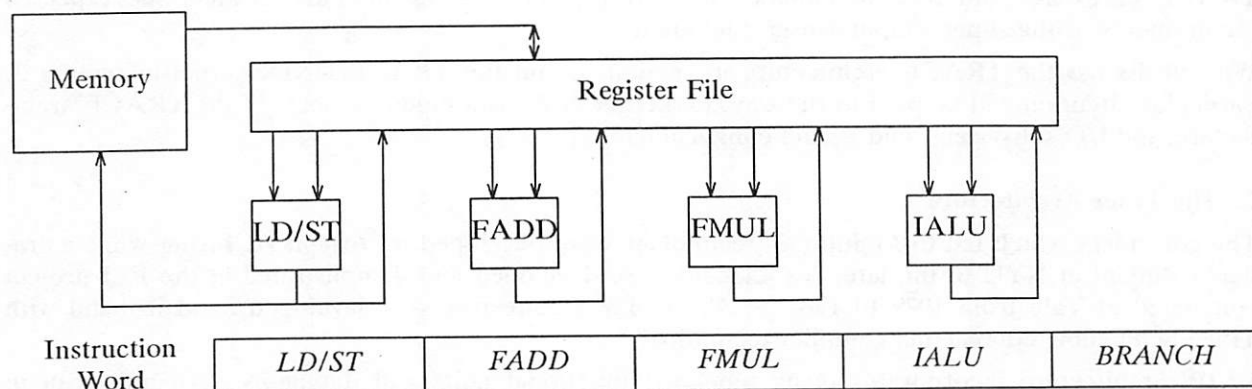


Figure 1: A Simple VLIW

A *scheduler* examines the relationships among operations of the program; then multiple *functional units* carry out independent computations simultaneously.

This fine-grained parallelism exists throughout all programs, and is independent of their high-level structure. As a result, overlapped execution has been a feature of nearly every high performance scientific and engineering computer built in the last twenty-five years. Examples include the CDC 6600 and all of its descendants; the Cray supercomputers; and the IBM STRETCH, 360/91, and descendants.

Overlapped execution has been universally successful and widely used, but the available speedups have been limited. Characteristics of programs have prevented computer designers from delivering larger speedups due to this low-level approach.

Gaining large benefits from overlapped execution requires that large groups of operations be candidates for overlap. As more and more operations are considered together for compaction, the performance improvement continues to grow. Large amounts of parallelism are found only when long streams of operations can be compacted.

However, a problem arises in trying to compact long streams. Programs are not straight-line streams of operations; they contain control flow statements, or *conditional jumps*. These pose a serious problem for scheduling. How can we overlap operations with prior conditional jumps?

```
      A = (B + C) * (D + E)
      IF (A .GT. 1.0E6) GOTO 5
      F = (G * H) + (X * Y)
5     CONTINUE
```

In this example, if the assignment of *F* and its computation occurred before the *IF* test, the program would produce incorrect results whenever *A* was greater than 1.0E6.

All previous efforts to overlap execution have overlapped only straight-line sections of code, or "basic blocks." Each conditional jump caused execution to serialize until its test resolved, and the scheduler could know which way to proceed.

Conditional jumps are found every five to eight operations in typical programs. This jump frequency, compiled with basic block compaction, has been the primary obstacle to very large performance gains from overlapped execution in scientific programs. When only small numbers of operations are candidates for overlapped execution, the gains from overlapped execution will be correspondingly small.

2.2. Trace Scheduling

Multiflow's Trace Scheduling compacting C and FORTRAN compilers overlap execution over long streams of code, going beyond many conditional jumps. The compilers use statistical information about program behavior, and a *compensation* technique, to perform aggressive compaction of long execution paths.

Trace Scheduling is carried out on one program module or subroutine at a time, after the program has been converted to an intermediate representation and after standard optimizations have been carried out.

Through heuristics, or by *profiling* a sample run of the application, the compiler obtains estimates of branch directions and loop trip counts. Using these estimates, long execution paths, or *traces*, are selected for compaction.

2.3. Trace Selection

Using loop trip count and branch probability information, the compiler selects the most frequent path, or *trace*, that the code will follow during execution. The path may include multiple conditional jumps.

This trace is then handed as a whole to a scheduler. The scheduler compacts operations into wide instruction words, taking into account data precedence and hardware resource constraints. These wide instruction words will be directly executed by Multiflow TRACE systems, using multiple functional units. Now, instead of five or eight operations which are candidates for scheduling, hundreds or thousands of operations may be candidates. Many opportunities for parallel execution will be present, and compaction will yield large speedups.

This large-scale overlapping moves operations in ways which could cause logical inconsistencies when a conditional branch goes the "less frequent" direction.

2.4. Compensation

Finding a method for handling these inconsistencies after compaction, without touching the compacted code, is the central innovation of Trace Scheduling. The compiler adjusts the flow graph of the remaining program to correct the scheduling-generated inconsistencies, and restore correctness for all execution paths. For example, if an operation above a conditional jump in the source winds up being scheduled below the jump, it is *copied* as part of the compensation for the jump. The copy is made only if there are computations in the code being jumped to which depend upon this operation.

The whole process then repeats. The next-most-likely execution path is chosen as a trace and handed to the code generator. This trace may include original operations and compensation code. It is compacted; new compensation code may be generated; and the process repeats, picking paths and compacting them until the entire program has been compiled.

2.5. Effectiveness

Trace Scheduling breaks the long-standing "conditional jump bottleneck" and finds parallelism throughout long streams of code, achieving order-of-magnitude speedups through compaction.

A number of conventional optimizations aid the Trace Scheduling process in finding parallelism. Automatic loop unrolling and automatic inline substitution of subroutines are both incorporated in Multiflow's compilers; the compiler heuristically determines the amount of unrolling and substitution, substantially increasing the parallelism that can be exploited.

2.6. Compiler Structure

Multiflow C and FORTRAN compilers are built around a common core. FORTRAN and C front ends generate a common intermediate language. The optimization and code-generation phases of the compilers are identical.

All FORTRAN and C programs receive the benefits of full optimization and Trace Scheduling compaction. System utilities and the operating system itself execute at high speed, because all C programs are fully optimized and compacted. All applications, whether written in FORTRAN or in C, benefit from Multiflow's compiler technology.

The Multiflow compilers perform extensive analysis and optimization of programs to improve performance. Optimization reduces run-time computation, and eliminates data dependency between operations, increasing usable parallelism. The optimizations performed include: induction variable simplification; common subexpression elimination; copy propagation; constant folding; dead code removal; register variable detection; loop invariant motion; variable renaming; inline substitution; loop unrolling.

Following intermediate-code optimization, Multiflow compilers *compact* the program: they schedule overlapped execution of program steps. Operations are scheduled using information about control flow, data dependencies, and hardware resources.

Control flow. Control flow analysis allows operation compaction beyond basic block boundaries. Statistical information, gathered from sample runs of the application or generated by compiler heuristics, guides the selection of long execution paths, or *traces*, for compaction. Each trace may contain

many conditional branches. The selection process begins with the most frequent execution path, compacting it for highest performance, then repeats, picking traces and compacting them until the entire program has been compiled.

Each trace is compacted as a whole. Instead of the small number of operations available within straight-line "basic blocks", hundreds or thousands of operations become candidates for overlap. Many opportunities for parallel execution are present, and compaction yields large speedups.

This large-scale overlapping moves operations in ways which could cause logical inconsistencies when the program branches off the chosen trace. Multiflow compilers automatically adjust the flow of the remaining program, adding small amounts of *compensation code* to correct these inconsistencies and ensure correctness for all execution paths.

Data dependencies. Data dependencies are managed while scheduling operations into wide instructions. Extensive analysis and optimization is performed to eliminate "surface" dependencies which result from the expression of the program, rather than from the algorithm itself.

Array references can pose special problems for compile-time data dependency analysis. Consider attempting to overlap operations from the fragment:

```
a[i] = (b + c) * (d + e);  
f = (a[j] * h) + (x * y);
```

Getting the best performance here requires compile-time analysis of the possible values of *i* and *j*, so as to be able to decide if the reference to *a[i]* can possibly refer to the same memory element as *a[j]*. If so, the memory load of *a[j]* must be scheduled after the store into *a[i]*, which will reduce parallelism somewhat.

Multiflow's Trace Scheduling compacting C and FORTRAN compilers perform exhaustive compile-time memory reference analysis. The compilers analyze the values which array index expressions can assume, and build symbolic derivations for their values in terms of local loop induction variables and invariant values. They then solve for whether or not the expressions can ever be equal.

Memory reference analysis and detailed compiler knowledge of the TRACE memory structure further allows compile-time management of memory banks. This provides high memory bandwidth via an interleaved, pipelined memory system without "stunt boxes" or hardware memory reference schedulers.

Hardware resources. Multiflow compilers incorporate a detailed model of the TRACE hardware which includes functional unit opcodes, pipeline depths, resource requirements, and datapath interconnect. The compilers completely control the operation of the hardware on a cycle-by-cycle basis, and manage system hardware resources such as buses, functional units, memory banks, and register write ports. Control and scheduling hardware has been replaced by compiler management.

2.7. TRACE Processor Organization

TRACE instructions initiate many operations simultaneously, using multiple Integer Arithmetic/Logical Units, multiple Floating-Point Units, and multiple Memory Units. Three models provide a range of performance, with successively larger instruction words and functional unit power.

- **TRACE 7/200:** Seven operations per instruction; 30 MFLOPs; 53 VLIW MIPs.
- **TRACE 14/200:** Fourteen operations per instruction; 60 MFLOPs; 107 VLIW MIPs.
- **TRACE 28/200:** Twenty-eight operations per instruction; 120 MFLOPs; 215 VLIW MIPs.

The entry-level TRACE 7/200 includes 160 32-bit data registers (used in pairs for 64-bit computations), with 0.9 Gigabyte per second bandwidth, handling seven independent computation steps in each 130 nanosecond instruction time. The most powerful TRACE 28/200 includes 640 general registers transferring over 3.6 Gigabytes per second, handling twenty-eight computation steps per

instruction. Register usage and functional unit assignment are managed by Multiflow's Trace Scheduling compilers.

The TRACE integer instruction set comprises over 80 operations, including arithmetic, logical, and compare operations; high performance primitives for 32-bit and 64-bit multiplication; conditional branching; shift, bit-reverse, extract, and merge operations for bit and byte field manipulations; and pipelined 32-bit and 64-bit load and store operations for referencing memory.

Each Integer Unit contains two Arithmetic/Logic Units units (ALU0 and ALU1) associated with a register bank of 64 general-purpose 32-bit registers. The register bank incorporates four read and four write ports and a bus-to-bus crossbar among its twelve bus ports.

The Floating-Point Unit was optimized for 64-bit IEEE standard 754 floating point computation; 32-bit format and computation is also supported. Like the Integer Unit, the Floating-Point Unit contains a bank of 64 general-purpose 32-bit registers with a bus-to-bus crossbar. 32-bit registers are used in pairs to hold 64-bit values. Functional units include a floating-point multiplier/divider (FMUL), a floating-point adder (FADD), and two integer ALUs. An additional register bank of 32 "Store" registers expands register bandwidth and improves memory reference performance.

The floating operation suite includes the integer opcodes, plus floating opcodes for addition, subtraction, multiplication, division, type conversion, and comparison.

Exception handling hardware provides several modes of operations, including full compliance with IEEE 754 exception processing.

Pipelined design techniques allow the multiplier, the adder, and the integer ALUs to initiate a new operation with every instruction regardless of the previous instruction.

Memory is virtually addressed by 32-bit byte pointers, with memory data layouts and formats compatible with industry-standard workstations, for easy program portability.

High CPU performance is balanced by large main memory capacity (up to 512 Megabytes), and by high sustained memory performance (up to 492 Megabytes per second performance). Unique cooperation between Multiflow's Trace Scheduling compilers and the TRACE hardware architecture allows the construction of a memory system which can sustain high bandwidth without the limitations of data caches or the costs of hardware memory-reference schedulers.

3. TRACE/UNIX

A VLIW may appear to be an odd sort of CPU to make into a virtual memory timesharing system. Indeed, the original designers of the ELI-512 expected their machine to be useful only as a number-crunching back-end processor [5]. The problems associated with making this heavily pipelined parallel machine capable of servicing interrupts seemed daunting enough, let alone all the rest: supporting virtual memory on a CPU without microcode, the incredible number of registers that would have to be context switched, extending the architecture and compiler to support systems code in addition to its numerical chores, not to mention the possibility that long instruction words might make all the utility programs consume gigabytes of disk space.

We've figured out ways around all of these problems, but it is natural to wonder why we built the TRACE to run 4.3BSD UNIX in the first place. The reason is simple: modern numerical applications programs do much more than perform floating point calculations. They make the usual demands of a system for disk, graphic, and terminal I/O, but they can make these demands at rates far exceeding those of "I/O intensive" systems programs. And scientific applications programmers have the same desires for reasonable and friendly programming environments that system programmers do. Fulfilling all these demands, particularly for performance, with a smoothly integrated front-end/back-end processor seemed difficult and unnecessary, so we built the operating system to run directly on the CPU.

3.1. Kernel Port Problems

The following sections describe the special problems encountered in porting UNIX to the TRACE VLIW, the solutions which were developed, and the enhancements which have been added along the way. TRACE/UNIX is ported from the BSD 4.3 VAX 11/780 source. All changes made by Multiflow are entirely internal to the UNIX kernel, preserving compatibility with the Berkeley distribution.

Some characteristics of the architecture were of particular concern during the process of kernel development:

- The hardware provides almost no support for the tasks involved in handling exceptions such as page faults, and in fact these tasks on the TRACE are unusually complex due to the large number of independent exception conditions which may arise on every machine cycle.
- The hardware has an unusually large number of general purpose registers, and there was some concern that save/restore overhead could be significant.
- The text size of programs compiled for the TRACE tends to be larger than that for a "typical" short instruction-word machine.
- The I/O system was radically different from the VAX's, and the balance between CPU-speed and I/O device speed was also quite different, so there was concern that radical redesign would be necessary to achieve reasonable throughput.
- Kernel debugging had to be done concurrently with hardware and compiler debugging, and with unique requirements for access to low-level hardware state information.

All of these concerns have been addressed in the TRACE implementation, with what we feel to be successful results.

3.2. Kernel Characteristics

The TRACE kernel is compiled with the TRACE Scheduling compiler, and so gets the same benefit from the technology as user code.

User processes may access a 4 gigabyte address space. The kernel runs within its own address space (ie, it has its own address space ID, described below). System calls from user processes are implemented via a trap sequence and context switch to this address space, which is all handled by code running in trap mode.

3.3. Support for a Multiple Process Environment

The TRACE includes all the architectural features needed to support a modern operating system: the instruction and data TLBs needed for virtual memory and mechanisms and constraints for dealing with exceptions.

The TRACE supports its multiuser operating system in the usual way. Appropriate protection modes and privileged instructions are provided so that the user process environment is maintained. All accesses to mapping hardware, I/O stimulus instructions, and the PSW are carefully protected. A limited set of traps to system mode are provided for system calls and breakpoints.

We were concerned about the effects of running multiple processes, and the overall impact that context switching would have on performance. Our goal was to support about as many users as would be comfortable on a large supermini but to support order-of-magnitude larger computations than current superminis could support.

Context switching is often considered to be simply the cost of saving and restoring registers. But the actual cost of a context switch also includes the interrupt time, scheduling overhead, and any penalty for cache purging and cold-start [6]. On many machines, the cost of purging the virtual address translation and instruction caches dominates register saving. The TRACE provides very large instruction and translation caches (see Sections 6.4 and 6.5), which are process tagged with an 8-bit "Address

Space ID", or ASID. No purging of the instruction cache or translation buffers is necessary on a context switch; caches must be purged only every 254 address space mapping changes, when the set of ASIDs overflows.

Updating the ASID registers is cheap, so the high available memory bandwidth in the system permits a complete context switch in 15 microseconds. This figure holds in any machine configuration, because usable memory bandwidth increases as the number of registers. This performance is comparable to other machines that are trying to support our number of users.

3.4. Interrupts

Interrupt handling is almost entirely conventional. There is a priority interrupt system, with maskable interrupts from each device. When an enabled interrupt request arrives, execution suspends, the processor changes state, and execution resumes at a "trap" address. Since the pipelines are self-draining, after the maximum pipe depth time, all of the state of the processor is either in general registers or in main memory; after several instruction cycles we enter C code to process the event.

3.5. Input/Output

Given an exposed architecture where the compiler knows about the machine resources being used throughout the system, it's difficult to allow I/O to "cycle steal" or otherwise share hardware resources on a fine-grained basis with program execution.

A memory-mapped I/O scheme would have required the CPU's memory interface to deal with devices with two distinct speeds: fast (to memory) and slow (to I/O devices). We chose not to implement our I/O this way. Instead, the CPU interacts with its devices through a surrogate called the I/O Processor (IOP). The IOP is based on an MC68010 with a multiported high bandwidth buffer memory and a "DMA engine" which can read and write blocks of main memory at half of peak memory bandwidth. The IOP interfaces to a VMEbus, a standard 32-bit asynchronous bus where the device controllers reside.

When the DMA engine wants to read or write main memory, it signals the global controller (GC). The GC suspends processor execution and allows pipelines to drain. The DMA engine then talks directly to memory at high speed; for example, 10 MB/s of I/O consumes only 4% of the machine's cycles in the largest CPU configuration. Execution resumes as soon as a burst of data has been transferred.

The I/O processor talks to the CPU using a bidirectional interrupt and a channel command protocol in main memory. Device drivers run on the I/O processor, a scheme which minimizes interrupts and CPU involvement in I/O operations. The IOP is also responsible for bringing the system up. A small operating system on the IOP, called MDX, supports execution of diagnostic and bootstrap programs.

3.6. Systems Code on a VLIW

The hundreds of thousands of lines of code which make up the UNIX kernel and utilities do not know they're running on a VLIW. One of our compilers is for the C language. Nearly all of the UNIX utilities, and a large chunk of the kernel, are written in portable C. (By actual count: 300 lines of assembly and 64K lines of C in the kernel; 1100 lines of assembly and 700 lines of C in the trap handlers.) The fact that our compiler performs exotic optimizations like Trace Scheduling and transforms the code into a parallel form is irrelevant. We compile these programs and they do what they're supposed to do; *grep* doesn't know it's stretching the frontiers of technology, it just greps along at a terrific rate.

Trace Scheduling was originally conceived for numerical applications; we expected to run into problems handling systems code. The systems code in UNIX differs in several respects from numerical code. Systems code makes pervasive use of pointers, which leads to more difficult compiler optimization problems. The code tends to have even smaller basic blocks than numerical code. And most

important, systems code has proportionately many more procedure calls than numerical code.

Pointers and small basic blocks have not been a problem. In fact, procedure call overhead seems to be the only issue that has required special attention. Performance on systems code is quite good (the C and Fortran compilers share a common back end).

The TRACE provides no special architectural support for procedure calls (other than the large memory bandwidth already built in). During the design, we considered several hardware mechanisms intended to minimize procedure call/return overhead, but none of them was both a clear performance win and clearly feasible. We decided to rely on the compiler to be clever with its use of registers and procedure inlining, and to develop a global register allocating linker, which builds a global call graph and minimizes register saves (currently in the works) [7]. We expect this work to be complete by the time of the conference presentation, and will report on it there.

When we initially debugged UNIX on the TRACE, we compiled without Trace Scheduling and loop unrolling; compiler heuristics for how much unrolling to perform had not yet been installed, and code grew unmanageably. Those heuristics are now in place, and their performance is remarkably good. Trace Scheduling and loop unrolling work well for a wide variety of systems code, including the kernel itself, without undue code growth.

This result surprised us somewhat; we hadn't anticipated as much improvement on systems code as we got. Good performance on systems code is very desirable, as it restrains the proportionate growth of operating system overhead that is usually encountered on a parallel machine. Unlike "coarse-grained" architectures where systems code runs on a single scalar unit (and can become a substantial bottleneck), we retain the same OS-to-user balance found on more traditional systems.

3.7. Code Size: Initial Results

The "no-op" fields of an instruction are not represented in main memory, so the object code size of a program is directly proportional to the number of operations in the compiled program. There are thus three components to consider when comparing VLIW code density to that of other architectures:

- the number of bits required within the instruction set to express a given operation; • the succinctness, or lack thereof, with which common high-level operations (like procedure call) can be expressed in the instruction set; and • the number of new operations introduced through compiler optimizations such as Trace Scheduling and loop unrolling.

The VLIW encoding of each operation is roughly on par with other RISC machines. It is a three address architecture, all loads and stores are explicit, and there is minimal instruction encoding. The code expansion per operation is probably around 30 – 50% when compared to a tightly encoded machine like the VAX or 68000. The variable-length main memory instruction encoding has an associated overhead of a few bits per operation, which coupled with main memory alignment constraints adds roughly an additional 5 – 10%.

Operations that cannot be initiated in a single instruction cycle are broken down into constituent sub-operations. These constituents are usually substituted inline, although certain operations such as the block register save and restore associated with procedure call are implemented via special subroutines. The overall code expansion due to this, as compared to a machine like the VAX that has an extensive library of microcoded "subroutines", is difficult to quantify, but is probably in the neighborhood of 10 – 20%.

The compiler performs an enormous number of optimizations, most of which reduce the number of operations in the program, but some of which increase the number of operations with the goal of increasing parallel execution. The three most notorious code-expanders are trace scheduling (which can produce compensation code), loop unrolling, and inline procedure substitution. All three of these are currently automatic and have been tuned to avoid undue code growth. These optimizations can increase the size of some small fragments of code by a large factor, but their overall effect seems to be to increase code size by a factor of around 30 – 60%, although the user can increase or

decrease these factors arbitrarily through the use of compiler switches.

Several large (100K -- 300K lines) FORTRAN programs have been built on the TRACE. After unrolling and trace scheduling, the code size is approximately 3 times larger than VAX object code (compiled with the VAX/VMS FORTRAN compiler).

The concern about code size led us to implement a shared-libraries facility very early in our UNIX development. This has substantially reduced the size of the UNIX utilities images. The Unix utilities consume approximately 20MB of disk space on a VAX, and approximately 60MB on our VLIW using shared libraries.

3.8. Hardware Management

3.8.1. "Microcode"

The TRACE systems have no microcode. All instructions are implemented directly in hardware, and are executed by the CPU in a uniform manner. However, the tasks often delegated to microcode in traditional architectures, such as virtual memory management, interrupt, exception, and process context switching, still remain and have to be done by something.

There are three main processor modes: user mode for user programs, system mode for the kernel, and trap mode for handling exceptions, traps, interrupts, and other special "events". Code written for execution in trap mode is just like code written for execution in user or system mode, and must obey precisely the same restrictions; the only difference is that in trap mode privileged operations are available for examining and manipulating low-level hardware state.

There is no complex hardware state machine associated with trap handling on the TRACE. If several events happen at once, the trap code is presented with the complete set, all at once. This means the software must perform event prioritization, and since the hardware never writes an exception stack frame, the trap code is also responsible for remembering all of the state. While this makes the software somewhat more complicated, it greatly simplifies the cost and complexity of the associated hardware.

When an event occurs, the trap code is entered through one of four vectors. The first vector is used if a machine check was detected; in this case the system is quickly halted, modifying as little state as possible, so the IOP can make an accurate analysis of what's wrong. The second vector is used only for asynchronous events, such as i/o or clock interrupts. The third vector is used if the only event is a system call (system calls are implemented via a variant of the breakpoint trap, of which several varieties are provided). And the last vector is used for events not categorized above, such as a TLB miss or floating exception, or for a combination of events precluding use of one of the other vectors; this represents the most difficult prioritization case. If the trap code is entered through the last vector it might have to deal with as many as 12 integer exceptions, 57 floating exceptions, 34 address translation faults, two i/o interrupts, a clock interrupt, and two programmable counter interrupts. If this sounds complex, imagine the cost to implement the arbitration and sequencing in hardware.

Considering all that it must do, the trap code is still quite compact. The source consists of 1100 lines of assembly and 700 lines of C code (less comments) and is common to the entire family of processors (7/200, 14/200, 28/200). The trap code text image itself consists of 2593 TRACE instructions, and is just 73K bytes in length.

The TRACE processor has a great deal of context, including 640 general purpose registers on the largest model. Saving and then restoring that much state can be expensive, even on a VLIW. The trap code has been implemented with fast paths for frequent events. When a trap occurs, the minimal amount of state for trap processing is saved. Some traps such as address translation faults and clock interrupts can be resolved quickly with sparing use of register resources. Other traps require a context switch entry into kernel mode; system calls fall into this category. Even in this case saving all of the state is avoided. The "call" to the kernel is treated as a normal procedure call; only under conditions possibly requiring later examination of the process state is all the state saved. This is achieved by backing out of the kernel to the trap code, executing the context save, and finally

reentering the kernel and continuing where processing left off. Receiving a signal is the mechanism that most commonly initiates this backing out activity.

3.8.2. Address Translation

For virtual memory translation the TRACE has separate instruction and data translation lookaside buffers (TLBs). Each TLB entry corresponds to one page of a virtual address space; given an 8K byte page size, the 4K element TLBs each address 32 megabytes of memory. Each entry contains an address-space ID (ASID) to which this entry belongs, the high order bits of the physical address, enough bits of the virtual address to verify match, and a "written" bit. One ASID (255) is reserved for the system; another must be used as the invalid ASID, as no "valid bit" is included in the entry.

Unlike the VAX, TLB entries are not flushed on context switch. The TLB is indexed using a hash of the virtual address and a set of kernel-maintained ASIDs for the current process. By choosing the ASIDs properly, TLB collision with the kernel (which has its own ASID) and with other processes can be avoided.

Separate ASIDs are maintained for text and data. Processes that share the same text segment use the same text ASID, reducing the possibility of collisions in the instruction TLB. Every process has a unique data ASID.

With the large TLB and the ability to scatter the translation of multiple process and kernel virtual addresses across the entire TLB, TLB reloading is reduced. Thus the task of page table entry lookup and reloading of the TLB on TLB miss (via a software trap) is not prohibitively expensive. Because reloading is being done by software, the implementation of the virtual memory architecture is very flexible, and instrumentation may be added at will to support debugging or performance monitoring. This has allowed extensive exploration of algorithms for reducing the number of software traps taken as a result of translation misses. Since the TLB management code is all written in C, this experimentation is exceptionally easy. We have even been able to experiment with per-process predictive strategies with relatively little effort.

To simplify operating system porting, the prototype virtual memory architecture corresponded very closely with that of the VAX. Except for the addition of code to invalidate TLB entries during paging or swapping (not needed on the VAX since TLB entries are flushed on context switch), very little modification to the TLB validation code as it existed in the Berkeley sources was needed.

3.9. Virtual Memory

Virtual memory issues include management of the user address space, and paging design. Enhancements in these areas include support for shared libraries and reduction of swap space usage.

3.9.1. Shared Libraries

In order to offset the code size expansion found in programs compiled for a VLIW, and the consequent increased need for disk space to store object code, a shared library facility has been added to the kernel. The main goal was object file size reduction; secondary objectives were established to make shared libraries flexible and easily managed, including the ability to modify public library code without recompilation or relinking of user programs, support for private user versions of library functions, version checking at run time, the ability to make multiple versions of the same library available concurrently, and support for user-created libraries. To assist in meeting some of these objectives, a general copy-on-write paging mechanism was also added. All the relevant kernel changes are completely transparent at the user level.

This section describes the implementation of shared libraries. Some user-interface issues are deliberately avoided in this discussion, as shared libraries have not yet been made available for use with the TRACE programming environment. Currently, a shared C library is supplied with systems, and almost all programs distributed with the TRACE use this library.

3.9.1.1. Process Types

A shared library exists as a user process, distinguished by a special magic number in its object file header. Thus, it is possible to determine what libraries are available, and their memory usage, by using standard utilities (e.g., *ps*). Programs which use shared libraries are also tagged with a special number, and in addition have a list of absolute pathnames of needed libraries in their headers. When a program is *exec'd*, the kernel maps code and data of each library specified in the header into the address space, and also checks a version number found with each pathname against the corresponding number for the running library. Both private and shared library text and data are demand-paged.

3.9.1.2. Address Space

The virtual address space of a process using a shared library consists of (1) shared library text area, (2) jump table area, (3) private text, (4) private data, (5) private stack, and (6) shared library data, in order of increasing virtual address. The library areas are sparse regions, within which addresses are allocated by library builders.

Every library consists of four "segments": jump table, text, initialized data, and uninitialized data (bss). (The library also has its own stack, which is not relevant to the user process). The library text is mapped somewhere in the first 256 megabytes of the user process space. This first 256 megabytes is globally shared among all processes using shared libraries. Processes which do not use shared libraries do not see this space (their private text starts at 0). Both using and non-using processes may freely coexist on a running system.

The jump table of a library is mapped copy-on-write into the first part of the *private* space of a process. The jump table consists of unconditional branch instructions whose targets are externally visible library functions. This table is actually contained in the object file of the library on disk, so that user program objects need not contain replicated copies for every library. During an *exec*, the jump table is mapped into private user space by simply copying page table entries, and marking them copy-on-write.

Shared library data sections are mapped as copy-on-write (initialized data) or zero-fill (uninitialized data) in a special area at the top of the user space, above the stack. The addresses are fixed at library link-time, and may not collide with the data of any other library being used by the process. (The same restriction applies to library text, although the jump tables make it easier to work around in that case).

3.9.1.3. Library Patching

Unlike some previous shared library designs that provide global jump tables, the TRACE version allows private user routines to be substituted for library routines without any modification to the library. Since all library calls go through the jump table, including internal library calls, the user may simply write over a jump table entry to make it point to private code (e.g., a private copy of *_doprnt()*). Since the jump table is mapped copy-on-write, this change is not seen by other user processes. Tools for automating this patching are not yet available.

3.9.1.4. Library Programming Environment

To use a shared library, the library path is specified on the command line to *ld* at link time; the symbol table of the library object is then accessed just as for an ordinary object file or library, but library text and data is of course not included in the output object module. With the exception of extra options to *ld*, linking to a shared library is not much different from linking to a non-shared version. A user can create a shared library using appropriate *ld* flags, but only the superuser can start a library process running.

Library text may be modified without changing the version number (and hence requiring re-linking of user code) if the jump table remains unchanged except for the possible addition of new entries at the end. This implies that changes which preserve the ordering, jump table offset, and semantics of

routines do not require a version number change.

When a new version of a library is created which requires a new version number, it is put in its own directory, and linked with a text load address which does not collide with the current running version. This library may then be started on a system which has an old version already running, and user programs linked with the new version will run; since the pathname of the new library is different, these programs will access the correct library. Versions of the C library are put in `/libsh/lib1.0/libc.a`, `/libsh/lib2.0/libc.a`, etc. This approach has proven to be highly useful, for example, when a test release of new user software, using a new library, is done on a running system.

3.9.1.5. Coexisting with the Hardware

The absolute load address of library text may be changed by simply re-linking the library. This is useful, for example, if an instruction cache or ITLB thrash is found to result from the current relative locations of the library text and jump table, or either of these and user text. Note that the jump table location may be moved around also within the fixed jump table area at the start of private user space, but this requires re-linking of user code.

Library code executes using the text ASID of the current user process. Thus, there may be multiple ITLB entries in existence at any given time for the same library pages. This potential waste of ITLB resources is somewhat ameliorated by the fact that shared user text segments use the same ASID. Future hardware enhancements to provide a special "shared" ASID for libraries are contemplated.

3.9.2. Paging System Enhancements

3.9.2.1. Copy-On-Write

Copy-on-write paging is implemented in the TRACE kernel, to support the shared library jump table semantics described above as well as reduce general unnecessary data copying overhead due to new process creation. The *vfork* implementation is also kept intact, retaining complete compatibility with programs which (unfortunately) depend upon its side-effects.

To support copy-on-write, two new fields are added to the *cmap* structure representing physical page frames: a reference count, and a saved page table entry. Also, a copy-on-write bit is added to page table entries. When a process forks, all valid in-core data page table entries of the parent are made read-only + copy-on-write in parent and child, and the reference count in each *cmap* structure is set to 2. Subsequent forks increment the reference count. On a page fault, the reference count is decremented and a copy of the page is made for the faulting process.

The main drawback of this simple scheme is that it does not provide any way for the kernel to find all processes which reference a copy-on-write page. This means, for example, that the pageout daemon must not put these pages on the free list, as it cannot find all page table entries which reference them. Swapping a process with copy-on-write pages is possible, however.

3.9.2.2. Page Replacement

The current TRACE implementation retains the standard 4.3 global clock page replacement algorithm. With a "batch-like" scientific programming job mix in which the number of very large processes is small, this has proven adequate to avoid thrashing; even the largest applications run so far on the TRACE rarely run out of physical memory or cause swapping activity by themselves.

3.9.2.3. Swap Allocation

The standard Berkeley UNIX implementation allocates swap space on disk for program text as well as data, even when the program consists of separate instruction and data spaces (the default). This approach makes it possible to handle the case where text pages may be modified when a process is being traced, for example when a process is being run under a debugger and breakpoints are set. It also yields a performance advantage for "sticky" text, since subsequent invocations of a program may

find text pages in the swap space rather than having to determine where they reside in the file system. Because of the text expansion factor inherent in VLIWs, it is undesirable to use disk swap space for read-only text which is not ever traced, if the performance advantage of doing so can be made irrelevant. The TRACE implementation therefore does not allocate swap space for text unless and until the process is traced. When a *ptrace()* is done on a process, swap space is allocated at that point for its entire text segment.

When a page fault occurs and the page table entry indicates the page is fill-on-demand from the filesystem (e.g., a text page not yet loaded), this entry is saved in the cmap structure before being overwritten (i.e., the disk block address of the page is remembered.) This mechanism is used to support fast paging of sticky text. When the last reference to a text segment is gone, the kernel structure for this text remains active ("sticky"); in the standard implementation, the page table for this text would be stored on disk and could be reloaded to reference text pages which had been stored in the swap space. The TRACE version puts all the page table entries for this text back to their fill-on-demand form, using the entries saved in the cmap structures, before writing the page table out to the swap space. When the text is again linked to a process, the text page table is ready to go with physical disk block addresses; it is not necessary to re-discover these addresses by examining the filesystem. Thus, the performance advantage of sticky text is retained, though swap space is not used.

3.9.3. Precise Timing

The coarse timing facilities generally provided by UNIX are neither consistent enough nor sufficiently accurate to meet the needs of high-performance VLIW users. We need more accurate accounting facilities in order to time programs which run for very short periods of time and to tune medium to large programs in which incremental changes sometimes produce slight, but measurable effects.

The TRACE provides several programmable counters which can measure cpu time, time spent in cache miss, etc. These counters are accurate down to the minor cycle time (65ns) and are accessible to the kernel and user code (such as profilers) as 64-bit values with support from the trap code. Rather than using the standard clock interrupt timing scheme, TRACE/UNIX uses one of these counters to measure cpu time, updating the resource usage slots in the U on each switch between user and kernel mode and on context switch, providing the accurate timing desired.

3.10. I/O Subsystem Design

The TRACE I/O system hardware comprises two parallel intelligent channels, each using a 68010 and 2 MB of local RAM, connected to CPU memory via a 123 MB/sec DMA path and a two-way doorbell mechanism. Each channel, or I/O processor (IOP), includes 0.5 MB of dual-ported RAM on an industry-standard, 20-slot VME bus.

This hardware was designed to support the classic approach of unburdening the CPU as much as possible by moving I/O processing onto programmable channels. However, there is a delicate balance which must be observed in such an attempt: it is easy to move too much I/O processing away from the far faster CPU, easing the CPU load, but hurting overall throughput.

For example, one could imagine moving a good portion of the file system to a disk-controlling IOP, in a plausible effort to both unload the CPU and "get the file system closer to the media." In most minisupercomputer systems today, this would probably be a mistake, given the computationally intensive nature of file system allocation and buffer management machinery. In our case, because the TRACE CPU is so wildly more effective at system code than a 680X0 processor of any class, and because the CPU can easily spare many megabytes of main memory for the system buffer cache, this would be a tragic mistake indeed.

We took the approach -- proven quite effective in practice -- of leaving the traditional UNIX I/O structures intact, but moving all "real-time" device controller interaction onto the I/O processor, which is optimized for exactly this task. On the CPU side, we built simple "generic" device drivers for disk, tape, terminal, Ethernet, line printer, etc., each of which defines a private channel protocol

with which it communicates with a set of corresponding IOP drivers. These private protocols are, in turn, layered on top of a common but extremely lightweight I/O request/response message-passing mechanism which insulates both the CPU and the IOP drivers from CPU/IOP DMA hardware specifics as well as device configuration details (more about the latter below).

These generic CPU drivers turned out to be wonderfully simple, small and fast. For example, the disk driver is about 100 lines of actual C code (*sans* comments), with 40 lines of that total in the strategy routine, and 30 lines in the I/O-done response routine. The generic tape driver is only slightly larger, because of the need to deal with user-visible tape transport state (handling end-of-tape detection, rewind status, etc.). These drivers consist mostly of packaging open, close, read and write requests into private protocol messages of the appropriate type, and unpackaging the responses that come back.

On the other hand, the IOP drivers bear the brunt of the nasty realities of dancing in the “real world” of VME controllers and their attached devices. Luckily, the bulk of these drivers is in exception handling and recovery, with much less code involved in the main task of making I/O happen. For example, one of the IOP disk drivers is 1000 lines of C code (again, not counting comments), with about 700 lines involved in initialization, watchdogging, and error recovery.

The I/O message-passing machinery alluded to above is the “connective tissue” of the whole I/O subsystem. Called the *ioc* module, it handles addressing, flow control, multiplexing and DMA hardware encapsulation of I/O requests between the CPU and I/O processor drivers. This rather formidable-sounding set of tasks turns out to be simple in practice, requiring only 100 lines of C code on the CPU side, and about 250 lines of code on each IOP (not counting the substantial initialization code, used once at boot time).

Another important goal we set for ourselves was a truly flexible device configuration scheme, permitting per-boot dynamic system reconfiguration. Except for the case of user-supplied device drivers, we wanted to enable the addition, reconfiguration, or removal of devices and controllers without requiring kernel recompilation or relinking. A secondary goal was to have only one kernel image and one IOP image (per release) that would suffice for all installations.

We’ve implemented this scheme using several cooperating mechanisms. At boot time, the master diagnostic IOP translates a human-readable and -editable system configuration file into *ioc* device mapping tables. These tables permit the *ioc* module to do dynamic translation of $\langle \text{major}, \text{minor} \rangle$ pairs into $\langle \text{IOP}, \text{device class}, \text{controller}, \text{device} \rangle$ addresses at each message request. This lower level of configurability, in turn, allows the */dev* directory to contain all possible devices, each with an immutable, canonical major/minor pair. Finally, the kernel image contains all generic device drivers, and the IOP image contains all supported particular VME controller/device drivers. IOP drivers are auto-configuring in the sense of probing for any controllers they might own; attempting to access a non-existent device that happens to be in the system configuration file will result only in an EXNIO error.

Additionally, the system configuration file contains the hostname, time zone, number of system buffers, and root, swap and dump device specifications, permitting us to completely free our users from the indignities of *config(8)*.

The cost of this configuration flexibility is a few kilobytes of unused driver code in the kernel and IOP images at any given installation, and one additional two-dimensional sparse matrix lookup per I/O request (for device mapping). Surprisingly, the initialization code for this entire scheme is not much larger than the auto-configuration code in standard Berkeley UNIX.

3.11. Debug

When attempting to port an operating system to hardware that is still under development, it is possible to cut weeks from the development schedule by careful planning and scheduling, so that implementation and debug take place in concert with hardware development.

A significant part of the TRACE software development went on before there was working hardware. An instruction-level simulator was implemented, which modelled the hardware exactly in all software-visible respects, and would detect and report all resource errors. This simulator has seen a great deal of use in compiler debug, even after the hardware was available, because it can model any hardware configuration, and supply exact details of program problems, some of which are not detected by the actual hardware, such as failure to restore registers upon return from procedure calls.

This simulator proved invaluable in writing the trap code. The basic trap code was debugged by simulating a test suite of 200 C programs running in a virtual address space, and faulting into trap mode to handle address translation faults and system calls. When the hardware became available, the trap code was already debugged, and was running on the hardware after two days of work by a team of two engineers.

When the hardware was first made available to the Operating Systems group, basic user mode instructions were functional. A short time later, virtual address translation through the data and instruction TLBs was working. However, the fault handling hardware was not functional until much later.

Fortunately a very large portion of the 4.3BSD kernel was debugged without being able to handle traps of any kind. The kernel and all user programs were compiled with system call "trampoline" code that would change instruction and data ASIDs. This code was used to emulate system call traps. A small amount of code was added to the kernel to detect when TLB entries could become invalid, for example during *fork*, *execve*, and *brk/sbrk* system calls, and context switches; in these cases, the page table entries were checked for page faults and TLBs would be reloaded before returning to the user process after the system call. By checking for faults before they could occur, no other modification was needed to the 4.3BSD kernel virtual memory code at this stage. Thus it was possible to exercise and debug most of the virtual memory code without hardware support. For example, it was possible to exercise the pageout daemon by recursively fork and exec'ing several processes with a restricted amount of memory. By the time the trap hardware was available, most user programs were ported and users could remotely log into the TRACE via Ethernet.

To aid debugging, a very simple resident debugger was added to the kernel that was controlled by the IOP. A cross-development version of *adb* was built that used a simple protocol based on UDP that would control the resident debugger via the IOP. Kernel breakpoints could be inserted with this debugger. (Note that in the early stage when traps were not implemented it was possible to emulate a kernel breakpoint by coding a branch to a breakpoint handler routine and moving the address of the next instruction to an unused register in a single VLIW instruction.) Emulating user mode breakpoint traps was deemed too difficult while the hardware did not support traps.

To examine the address space of any loaded process, *adb* was extended to interpret process page tables via */dev/mem* or remotely via the IOP. Once hardware traps were implemented this extension proved to be useful in tracking down bugs in the virtual memory paging code. For on-site diagnosis, a version of *adb* for the TRACE runs on the IOP as a process under MDX. The *ps* utility is also available on MDX, to show the state of processes on the TRACE. Having MDX-based standard tools like *ps* is useful, for example, when the kernel running on the TRACE has been stopped or the machine has crashed.

4. Conclusions

UNIX has been running on the TRACE and supporting its own development for some time. The principal advantage of the Trace Scheduling/VLIW parallel processing technology is that it is largely transparent to its clients. Thus, most of the challenging problems in developing an operating system and programming environment for the TRACE come not from its VLIW nature but from our intention to make the system into a first rate environment for high performance engineering and scientific computation.

5. References

- [1] Computer Systems Research Group, "4.3 Berkeley Software Distribution, Virtual VAX-11 Version," *Department of Electrical Engineering and Computer Science*, University of California, Berkeley (April, 1986).
- [2] Joseph A. Fisher, "The Optimization of Horizontal Microcode Within and Beyond Basic Blocks: An Application of Processor Scheduling with Resources," *Technical Report COO-3077-161*, Courant Mathematics and Computing Laboratory, New York University (October 1979).
- [3] John R. Ellis, Joseph A. Fisher, John C. Ruttenberg, and Alexandru Nicolau, "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM SIGPLAN Notices (June 1984).
- [4] John R. Ellis, *Bulldog: A Compiler for VLIW Architectures*, MIT Press, Cambridge, Mass (1986).
- [5] Joseph A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *Proceedings of the 10th Symposium on Computer Architectures*, pp. 140-150, IEEE (June, 1983).
- [6] Douglas W. Clark and Joel S. Elmer, "Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement," *ACM Transactions on Computer Systems* 3(1), pp. 31-62 (February 1985).
- [7] David W. Wall, "Global Register Allocation at Link Time," *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, ACM SIGPLAN Notices (July 1986).

UNIX without the Superuser

M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer,
V.D. Gligor¹, W.D. Jiang, A. Johri, G.L. Luckenbaugh, N. Vasudevan

IBM Federal Systems Division
708 Quince Orchard Road
Gaithersburg, Maryland 20878

ABSTRACT

We consider the problem of enforcing the *Least Privilege Principle* (LPP) from a UNIX²-based operating system, Secure Xenix³. The LPP requires that you operate with the minimum set of privileges to accomplish your work. To enforce the LPP, Secure Xenix (1) partitions the UNIX superuser privilege into about three dozen privileges based on privileged system calls and privileged options of nonprivileged system calls, and recasts former set-UID root programs to the minimal necessary subset of these privileges; and (2) partitions the UNIX superuser role into five separate privileged roles and introduces a flexible table-driven framework for the elaboration of other (or the merging of existing) privileged roles, each with minimal privilege. We discuss design and implementation decisions of and experience with this privilege regime. We conclude that transition to the resulting system is doable, is desirable for tight security, suffices for LPP enforcement, and should not be precluded by emerging UNIX standards like POSIX⁴.

Key Words and Phrases: security, operating system security, DoD computing security, least privilege principle, UNIX, superuser, privilege.

Disclaimer: The work reported herein is part of a research project. No IBM product commitment is made or implied.

1. Introduction

Problem Statement. To enforce the LPP in a UNIX-based operating system, we consider the problem of eliminating the superuser role and privilege in multi-user mode, maintaining compatibility of nonprivileged programs with a standard like POSIX [POSIX], and influencing UNIX security standards (e.g., regarding privilege). The UNIX superuser, both as administrative role and privilege, concentrates too much authority and power in one place; penetrating this role (privilege) makes the whole system vulnerable. With UNIX, privilege is all or nothing; finer privilege granularity than this is more desirable. Thus, we revisit the notion of privilege in UNIX, seeking a design that is more granular, more flexible, and eliminates ties to a user ID. The superuser (or omnipotent privilege) is still needed in (say, single-user) maintenance mode for functions like installing system releases, fixing problems, and maintaining the system configuration, but is completely replaced during normal system operation.

¹ V.D. Gligor's permanent address is:
Dept. of Electrical Engr., Univ. of Maryland, College Park, Maryland 20742.

² UNIX is a registered trademark of AT&T.

³ Xenix is a trademark of Microsoft Corporation.

⁴ POSIX is a trademark of IEEE.

The Least Privilege Principle is key to this exercise. In [SS] (also see [S]), Saltzer and Schroeder give the following explanation of this fundamental principle of information protection.

"Least Privilege. Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted or improper uses of privilege are less likely to occur. Thus, if a question arises related to misuse of a privilege, the number of programs that must be audited is minimized. Put another way, if a mechanism can provide "firewalls," the principle of least privilege provides a rationale for where to install the firewalls. The military security rule of "need-to-know" is an example of this principle."

Our Solution. As a research project, we have developed (designed and implemented) a secure version of Xenix called *Secure Xenix* [G+] that runs on an IBM PC/AT workstation. To enforce the LPP, Secure Xenix (1) associates no special and no preassigned privileges with UID 0 (*superuser*, *root*) nor any other UID; (2) prevents login as root; (3) has no *su*; (4) partitions the superuser role into five separate privileged roles (four with UNIX-style groups) and introduces a table-driven framework that allows the elaboration of other (or the merging of existing) privileged roles; (5) partitions the superuser privilege into about three dozen privileges based on privileged system calls and privileged options of nonprivileged system calls; (6) recasts former set-UID root programs to use the minimal necessary subset of privileges; and (7) offers "privilege bracketing" (for delineating the use of privilege) inside programs. This paper tells the design, implementation, and (some of the) experience story of this privilege regime.

Our solution has the following advantages. First, it applies the LPP to both the superuser role and privilege of UNIX, and maintains compatibility. Second, it identifies and separates privileged administrative roles, and avoids the per-role password management problem. Third, it introduces a flexible, table-driven framework for privileged roles. Here, "flexible" means that we can easily introduce, merge, and redefine roles; and we can easily define and enforce a power hierarchy among roles. Fourth, it integrates privileged roles with a B3 "trusted path" [TCSEC]. Regarding the contribution of this paper, we know of no other work that combines all these advantages:

- UNIX
- LPP
- compatibility
- granular privilege regime
 - privilege vectors
 - privilege bracketing
- separate administrative roles
 - privileged groups
 - no per-role password management problem
 - flexible, table-driven role framework
 - power hierarchy among roles
 - integrated with trusted path

Since Xenix is a representative version of the UNIX operating system with regard to privilege mechanism, this work can apply to any such monolithic superuser privilege mechanism. There are degrees of LPP enforcement. *All or nothing* (e.g., UNIX) is better than *all only* (e.g., PC-DOS); finer privilege granularity may be better than coarse privilege granularity, to some point of diminishing returns for design tradeoffs.

Extant Work. We know of three studies that pertain to our work. Bishop [B] considers the management of the superuser account and privileges in UNIX, and discusses the partitioning of the superuser role. For LINUS IV, also UNIX-based, Kramer [K] partitions the superuser role into three separate roles (Security Officer,

Operator, and Administrator), and provides per-role restrictive shells. The previous version of Secure Xenix [G+] partitions the superuser role into five privileged roles, associates a pseudo-user with each role, and provides per-role restrictive login shells. None of these designs solves the per-role password management problem, nor provides a flexible table-driven role framework, nor addresses the power hierarchy among roles, nor partitions the superuser privilege, nor integrates privileged roles with a trusted path.

Structure of This Paper. The rest of this paper has six sections. Section 2 explains how Secure Xenix partitions the superuser privilege. Section 3 reviews previous approaches to partitioning the superuser role. Section 4 explains how Secure Xenix now partitions the superuser role. Section 5 presents a framework for privileged administrative roles. Section 6 covers some of our early experience with this privilege regime, and Section 7 summarizes our conclusions.

2. Privilege Vectors

2.1. Why?

Why consider changing the superuser privilege mechanism? The answer is that, the superuser privilege may offer too coarse a privilege granularity for "tight" security. By *granularity* we mean the relative fineness or coarseness by which a mechanism can be controlled or adjusted. With a superuser-based UNIX, privilege is all or nothing, and *all* is only a first approximation to *least*.

To understand our privilege granularity better, we now review the privilege (or authorization) policy of the superuser privilege.

UNIX has three privilege classes of system calls: privileged, nonprivileged with privileged options, and nonprivileged with no privileged options. As a specific example for Secure Xenix, Figure 1 lists the privileged system calls, and Figure 2 lists the nonprivileged system calls with privileged options. In a superuser-style kernel, "privilege" means that the effective UID of the process is 0 (i.e., root or superuser). Sometimes it means that the process has the same privilege as the owner of the object. For example, *mount()*, which mounts a file system, is a privileged system call; only a process with an effective UID of 0 can execute *mount()* successfully. As another example, *chown()*, which changes the owner and group of a file, is a nonprivileged system call with privileged options; the root can change the owner and group of any file, while a non-root user must be the current owner to change either the owner or group of a file.

To enforce the LPP in a superuser-style regime, we can minimize privileged user-level code that runs with UID of 0 (set-UID 0 programs), divide user-level privileged code into small routines that support a single privileged function, and for trusted processes grant access to only a subset of privileged programs that are necessary.

In general, coarse privilege granularity characterizes problems with enforcing the LPP with a superuser regime. We cannot grant access to only a subset of the privileged system calls, or to only a subset of the privileged options of a system call. The division of privileged code into small support routines may be difficult for some large trusted processes such as a secure tape archiver (e.g., *star* in Secure Xenix), and *login*. In addition, access policies for privileged roles tend to be hard-coded into programs, an undesirable practice.

With this in mind, the goals of the Secure Xenix privilege mechanism are finer control over access to privileged system calls and privileged options, ability to configure the access policies of privileged roles to the needs of a particular installation with configurable privilege binding, and compatibility of nonprivileged programs.

2.2. Privileges

Secure Xenix does not associate privilege with a user ID; instead, it associates privileges with running processes and executable files. Each process has an associated privilege set, a (possibly null) subset of all privileges, represented by a bit-vector where each bit has a fixed privilege interpretation that specifies its privileges. Before a process can execute a privileged operation in the kernel, code in the kernel checks that the process has the corresponding privilege.

On Secure Xenix, file `/usr/include/sys/s_priv.h` defines a privilege vector, `typedef priv_t`, as an array of two unsigned long (32-bit) integers, which can represent up to 64 privileges. Associated with each defined bit position is a manifest constant with prefix `"PRIV_"` and an associated but fixed privilege interpretation. The suffix of each of these manifest constants identifies the privilege. The set of privileges partition and replace the superuser privilege. For convenience, we typically omit the `PRIV_` prefix when we discuss or specify privileges. For example, we say privilege `CHOWN` rather than `PRIV_CHOWN`.

File `s_priv.h` defines about three dozen privileges. Bits 0 and 1 correspond to privileges `MAC_EXEMPT` and `DAC_EXEMPT`. A process with privilege `MAC_EXEMPT` is exempt from mandatory access control checks (e.g., see `[TCSEC,G+]`). Likewise, a process with privilege `DAC_EXEMPT` is exempt from discretionary access control checks (e.g., see `[TCSEC,G+]`). Aside from `MAINT_MODE`, discussed below, the remaining privilege bits generally correspond to a privileged system call or the privileged options of a nonprivileged system call. In some cases a single privilege may be used with more than one system call. For example, `ACL[G+]` overloads the privileged options of `ACL` system calls; `LINK_DIR` is the privilege to link or unlink directories; `MOUNT` is the privilege to call either the `mount()` or `umount()` system call; and `SETUID` is the privilege to call either `setuid()` or `setgid()` as the superuser did. In these cases we saw no need for finer granularity and so adopted these compound privileges. Likewise, for nonprivileged system calls with more than one privileged option, we have not yet found a need to distinguish between the options with separate privileges.

Except for privilege `MAINT_MODE`, bit 63, we identified each of these privileges by locating all checks in the vanilla Xenix kernel of the UID (`u.u_uid`) with 0 and all `suser()` calls. Almost every such check corresponds to either a privileged system call or a privileged option of a nonprivileged system call. Therefore, we named these privileges after the associated system call. The exceptions are privileges `MAC_EXEMPT`, `DAC_EXEMPT`, and `MAINT_MODE`. Privilege `MAINT_MODE` identifies if a process is in the single-user maintenance mode; the kernel communicates this information to `init`. To set a file label to the special "wild card" security label `[G+]`, which makes the file (e.g., `/dev/null`) `MAC` exempt, the kernel checks for privilege `MAINT_MODE`.

Our privilege vector design philosophy restricts privileges to only those privileges checked by the kernel; no privilege bit corresponds to a trusted process (e.g., the privilege to change a password), an intentional simplifying decision that makes the set of privileges more stable. We need only modify the set of privileges if we add a new privileged system call or privileged option (a rare event), or port our privilege mechanism to another UNIX kernel with additional privileges that do not map onto existing ones. Adding a new trusted process has no impact on the set of privileges. Because almost all privileges correspond to system calls, it is easy to identify those privileges a trusted process needs by scrutinizing the system calls it invokes and, for `MAC` and `DAC` exemption, by scrutinizing the objects it handles.

On Secure Xenix we have increased the size of an on-disk i-node from 64 to 128 bytes. Among other security relevant file attributes, the extra space holds a privilege vector. Associated with each i-node is a privilege vector (`fpriv`, "f" for file), which is all zeros at file creation and at open-file-for-write-access, until it is set otherwise by the new system call with this purpose.

2.3. System Calls

Associated with each process are three privilege vectors: the inherited privilege vector (*ipriv*), the maximum privilege vector (*mpriv*) and the effective privilege vector (*epriv*). The *ipriv* is the *epriv* immediately prior to the most recent *exec()*. The *mpriv* indicates the maximum set of privileges that a process can have. The *epriv* indicates which privileges the process currently has; privilege checks are done against this privilege vector. Both *epriv* and *mpriv* are initialized to the union of the *epriv* of the process before its last *exec()* and the *fpriv* of the file that was *exec'd*. The two (*epriv*, *mpriv*) may differ because a process can temporarily drop privileges for privilege bracketing. On *fork()*, the child process inherits the three privilege vectors of the parent. The need for *ipriv* solves a privilege problem with the *mkdir* command on Secure Xenix and the *access()* system call. The *mkdir* command calls *access()* on behalf of its client, not itself; *mkdir* wants to know if its client has access to make a directory. In a superuser-style UNIX kernel, system call *access()* checks if the process identified by the real UID and GID, not the effective UID and GID, has access. When not in set-UID (set-GID) mode, real and effective IDs are the same. When in set-UID (set-GID) mode, real and effective IDs are different, and the set-UID (set-GID) program wants to ask an access question on behalf of its client as identified by the real IDs. In the Secure Xenix kernel, system call *access()* temporarily uses the inherited privilege vector for its calls to a function that checks mandatory and discretionary access to an object.

Secure Xenix has four new system calls for privilege vectors:

<i>getfpriv(path, priv)</i>	-	Get file privileges.
<i>setfpriv(path, priv)</i>	-	Set file privileges.
<i>getppriv(ipriv, mpriv, epriv)</i>	-	Get process privileges.
<i>setppriv(cmd, priv)</i>	-	Set process privileges.

System calls *getfpriv()* and *setfpriv()* respectively get (read) and set (write) the privilege vector of a file. System calls *getppriv()* and *setppriv()* respectively get (read) and set (write) the privilege vector of a process. Of these, only *setfpriv()* is privileged. System call *setppriv()* has three *cmd* values: *drop* to permanently drop privileges from both *mpriv* and *epriv*, *lapse* to temporarily drop privileges from *epriv*, and *acquire* to reacquire lapsed privileges from *mpriv* to *epriv*. The *priv* argument to *setppriv()* is ANDed with *mpriv* before it is used.

A process inherits privileges on *fork()* and *exec()*. On *exec()* the process also picks up the privileges in *epriv* and *mpriv* of the file that was *exec'd*.

In general inside the kernel, a process cannot execute a privileged operation unless the corresponding *epriv* bit is set. What was a UID test against 0 in a superuser-style kernel is now replaced in the Secure Xenix kernel with a call to a function that tests *epriv*, with a few exceptions. As noted above, *access()* uses *ipriv* not *epriv*. Also, *kill()* uses *mpriv* to make sure for safety that the killer process does not have less privilege than the victim process, except if the killer has *PRIV_KILL*.

An unresolved compatibility issue is whether the *setuid()* system call should as a side effect change *epriv* and perhaps other privilege vectors. Currently in our implementation it does not do so. The problem is with old set-UID root programs that use *setuid(nonzero uid)* as a method of restricting their (ordinary UNIX-style) privileges, as opposed to programs that use *setuid()* just to change their DAC access rights. We are considering several proposals.

3. Previous Approaches to Partitioning the Superuser Role

Partitioning the superuser role is not a new idea. Several efforts have divided UNIX superuser authority into separate special roles, and provided restrictive user interfaces (e.g., "login shells") for these roles; LINUS IV [K] and the previous version of Secure Xenix [B+] are representative of this approach. In contrast, Bishop [B] focuses on managing the existing superuser account.

3.1. Managing Superuser Privileges

Bishop [B] considers the necessity and management of the superuser account. He asks whether the superuser account is necessary even if one were to remove privileged system calls as special cases and reorganize the entire system accordingly. He concludes that the superuser account is necessary for two reasons: to terminate processes in an emergency, and to read all files or special files (i.e., device files) for system activities like file system backup. He notes that "all other actions a superuser might perform could be safely delegated to other users and groups, since none requires the protection mechanisms built into UNIX to be overridden."

He continues by saying, "But it is possible to have the kernel provide a compartmentalized security scheme in which each of several special accounts has exactly one property or ability of the root account; for example, the account authorized to terminate any process on the system would not be able to read other users' protected files. This limits the damage that someone using such an account can do. Implementing this would of course require massive changes to the kernel. Given that this is not acceptable for various reasons, we shall have to act on the assumption that a superuser account need exist." Our experience contradicts this, as we shall show below.

Bishop considers "three ways to organize the root account: have just one account; have several accounts each with the same powers; have several accounts with the powers of root divided among them." The conclusion is that the second method has no advantages over the first, and the third method has technical advantages over the first, but it suffers from the password management problem. The third method splits the root account into other less powerful accounts with per account passwords. Functions of these less powerful accounts include:

- accounting
- maintaining user accounts,
- rebooting and halting the system,
- backing up and restoring files,
- editing system files,
- daemons,
- owning system binaries and their sources, and
- handling emergencies.

Bishop states that there are other ways of dealing with the superuser privileges not discussed in [B], such as better exploiting the UNIX group mechanism, and using the least privilege concept.

3.2. LINUS IV

LINUS IV [K] splits the single superuser of UNIX into three separate users: the *Security Officer*, the *Operator*, and the *Administrator*. The Security Officer maintains the system security and manages auditing; the Operator takes disk backups, starts and halts the system, runs integrity checks, and fixes some integrity problems; and the Administrator issues and deletes user accounts in conjunction with the Security Officer. These special users are limited as to what they can do; they cannot run a standard UNIX command interpreter (a shell). Instead, these special users have a restrictive interface with a few "canned" procedures. On LINUS IV, you cannot login as root, and you cannot *su* to root.

3.3. Secure Xenix, Previous Design

Secure Xenix identifies the following privileged roles:

TSP	(Trusted System Programmer),
SSA	(System Security Administrator),
Auditor,	
SO	(Secure Operator), and
AA	(Accounting Administrator).

TSP Functions. The TSP is, in essence, the superuser restricted to single-user maintenance mode. The TSP (1) defines the node (site) identifier; (2) establishes and maintains the system configuration; (3) installs the system; (4) defines accounts and

passwords for SSAs and Auditors; (5) customizes system security tables; (6) manages recovery after system crashes; and (7) as necessary evolves the TCB (Trusted Computing Base) [TCSEC].

SSA Functions. The SSA is responsible for the security state. The SSA (1) creates, changes attributes of, and deletes user accounts and groups; (2) manages the password mechanism; (3) maintains the tables that define security labels; (4) manages the security labels on file systems and devices; (5) performs trusted upgrading and downgrading; (6) defines security labels for unlabeled imported files; and (7) in an emergency terminates rogue or runaway processes. (Item (7) can be an SO responsibility.)

Auditor Functions. The Auditor (1) enables and disables auditing; (2) determines auditing selectivity; (3) manages the audit trail; and (4) runs various report generation tools on the audit trail.

SO Functions. The SO (1) manages the system printer(s); (2) performs various file system tasks (mount, unmount, backup); and (3) can shut down the system.

AA Functions. The AA is responsible for managing the accounting trails and producing accounting reports of system usage. The AA (1) enables and disables the accounting trail; and (2) runs various report generation tools on the accounting trail.

The previous version of Secure Xenix [G+] had a different pseudo-user for each privileged role (SSA, Auditor, SO, and AA), and a restricted login shell for each of these roles. This design is problematical in practice due to per-role password management. We want to allow many possibilities for mapping users to roles: one user per role, many users per role, and many roles per user. If one user has many roles and we have per-role pseudo-users, then that user needs to remember many passwords, each of which ages. If many users have one role, then whenever one role member changes a pseudo-user password due to aging, then she must tell the other same role members. To revoke privileges to a role with per-role pseudo-users, we must change the role password and tell the other same role members. Furthermore, to distinguish different SSAs, for example, for auditing purposes, an SSA would need to identify herself, say with her real password.

3.4. General Problems of Partitioning the Superuser Role

In general, examples of the approach of partitioning only the superuser role tend to have several common problems. First, the privileged roles are fixed (hard-wired) at system design time, and there is no flexibility in easily reconfiguring them (defining new roles, merging existing roles). Second, with per-role pseudo-users, this approach suffers from the password management problem. Third, per pseudo-user login shells are typically inflexible in that it is not easy to add and subtract commands. Fourth, the restrictive login shells are set-UID root programs, which can potentially provide all privileges; there is no way to fire-wall the privileges of such login shells.

4. Partitioning the Superuser Role, Revisited

Our current approach partitions the superuser role into the same five privileged roles (TSP, SSA, Auditor, SO, AA) as the previous version of Secure Xenix. Two differences, though, are the power hierarchy among roles, and the representation of roles as groups. Another difference is that, with privilege vectors, administrative programs need not have all privileges, only what they need.

4.1. Power Hierarchy among Roles

Figure 3 shows the power hierarchy among Secure Xenix roles, represented as a directed acyclic graph with transitively implied arcs omitted. In the power hierarchy graph, an arc (A, B) means that tail role A "dominates" (or "has more power than") head role B. The TSP has the most power, and the TSP dominates the SSA and Auditor roles. The SSA and Auditor roles are independent. The SSA dominates the SO and AA and User roles. In Figure 3, User means an ordinary user. By transitivity, the

TSP dominates all other roles in Figure 3. Note that while dominance implies greater relative power, the converse is not true; for example, the Auditor has a powerful role, but does not actually dominate any other group.

An interesting problem arises from the power hierarchy, and from the SSA role in particular. The TSP defines user accounts for each SSA and each Auditor. The SSA defines user accounts for each SO and AA and User. The SSA can change passwords of users subject to some restrictions. An SSA can change her own password, but an SSA should not be able to change the password of an Auditor or another SSA. Otherwise, one mischievous SSA can masquerade as another SSA to redirect blame, or as an Auditor to erase or nonrecord events. Similarly, an SSA should not be able to change an attribute (e.g., owner, security label) of an object owned by the TSP or another SSA or an Auditor. The problem is: How can we implement these restrictions in a way that does not hard-code the names of roles, a bad practice, in programs like the password program?

To solve this problem, Secure Xenix explicitly represents the power hierarchy in the Group Table `/etc/security/s_group` (each group has a "group label"), and provides a library function to check the privilege of a subject (user and group) to change an attribute of an object (user or group or file). For example, the password program calls this function to determine if the caller can change the password of someone else.

4.2. Representing Roles

Secure Xenix represents roles SSA, Auditor, SO, and AA with privileged administrative groups *ssa*, *audit*, *so*, and *aa* respectively. To make someone an SSA, for example, a TSP simply adds that user to the *ssa* group.

The login program of Secure Xenix prompts for name, password, group, and security label. If I am an SSA and I want to login as an SSA, then I login with my name "matthew", my password, and the group "ssa". When I login, though, I get my standard login shell (e.g., `/bin/msh`). With no per-role pseudo-users, no *su* and no *newgrp*, I need know no other passwords! We could have changed login to recognize a privileged administrative group at login time, and to run a restricted shell for each such group. Instead, we adopted another design, which the next section describes.

5. A Framework for Privileged Roles

The previous version of Secure Xenix [G+] had (1) a trusted communication path mechanism, the Secure Attention Key (SAK), that, when invoked after login, would execute a trusted shell for ordinary users to change passwords and perform secure diskette I/O; and (2) separate trusted login shells per role. This design has "early decision binding" problems. First, since commands in these shells were implemented with control flow, recompilation and reinstallation is necessary to add or subtract commands, and even to change menu names and explanations. Experimenting with different commands, an important design and evolution activity, is at best cumbersome. Second, this design tends to hard-code the names of roles in various places, like the names of the per role shells and prompts. We would rather provide a default set of roles that a TSP can customize easily, say by merging existing default roles or by introducing new roles. Third, creating, fixing, and testing commands for each of these different shells requires knowledge of how each shell works. We would prefer a divide-and-conquer framework that allows us to focus on individual commands.

To solve these problems, we replaced the previous shells with a table-driven Trusted Shell that integrates commands for both ordinary and administrative users. The new design still uses the same trusted communication path mechanism.

The Trusted Shell (*tsh*) is a table-driven command interpreter; Command Table `/etc/security/s_cmd` (*s_cmd*) drives the interpreter. The subsections below cover *tsh* invocation, *s_cmd* syntax and semantics, and *s_cmd* population.

5.1. Trusted Shell Invocation

There are two ways to invoke *tsh*: with the SAK, and by executing `/bin/tsh` directly. The SAK consists of two `^z` (read "control z") characters in quick succession, less than one second between the two `^z` characters. (This is reconfigurable at system generation time.)

Invoking *tsh* with the SAK produces a *trusted communication path* (hence the adjective "Trusted" for Trusted Shell), a mechanism by which a person at a Secure Xenix terminal or the console can communicate directly with the Trusted Computing Base (TCB), and cannot be imitated by untrusted software. The Trusted Shell and the Command Table (and the SAK) are part of the TCB because they are part of the trusted path mechanism. If you are "talking to" *getty* or *login* when you press the SAK, then the SAK causes *init* to spawn a new *getty*, which reissues a login prompt. After login, however, when you press the SAK, you see the following output:

```
** Trusted Communication Path
** Type ? for a "tsh" command menu.
tsh#
```

Executing *tsh* by pressing the SAK, the intended invocation, is not the same as executing *tsh* by invoking `/bin/tsh` directly, which is harmless but sometimes useful as a login shell. When you invoke the Trusted Shell directly, then you see the following output:

```
** Not a Trusted Communication Path!
** Type ^z^z for a trusted communication path.
** Type ? for a "tsh" command menu.
tsh#
```

When you press the SAK, *init* writes a special value in file `/etc/utmp` to indicate that you have entered the *tsh* via the SAK, then *init* executes `/bin/tsh`. Some *tsh* commands contain a "guard" function that prevents the command from being executed outside the *tsh*; this guard examines file `/etc/utmp`.

The *tsh* is a restricted shell; you cannot execute everything that you have normal execute access to, say, from `/bin/sh` or `/bin/csh`. The only commands that you can execute in *tsh* are those listed in a menu, which you can see when you type ? (a question mark). The Command Table is the only source of commands for a *tsh* menu.

5.2. Command Table Syntax and Semantics

The current Secure Xenix Command Table contains 48 entries. Table *s_cmd* can have both comment and non-comment lines. A comment line in *s_cmd* begins with a `#` in column one. Each non-comment line in *s_cmd* is an entry

command:type:explanation:path:menus

that represents a command and its attributes, and has five colon-separated fields:

command	- command name (for a tsh menu)
type	- command type (see below)
explanation	- command explanation (for a tsh menu)
path	- pathname of trusted command program (when type is p), a predefined built-in name (when type is b), pathname to check execute access (when type is s)
menus	- a nonempty sublist of [tsh,ssa,so,audit,aa] or other TSP-identified privileged groups

There are three command types:

- p - pathname command (e.g., /bin/passwd)
- b - built-in command (e.g., cd)
- s - state transition to an administrative state (e.g., ssa)

Here are some examples of Command Table entries.

```
?:b:print this help menu::tsh,ssa,so,audit,aa
cd:b:change the current directory::tsh
ssa:s:ssa commands:/usr/security/bin/c_chlabel:tsh
star:p:secure tar:/bin/star:tsh,so
^d:b:(control-d) leave tsh::tsh,ssa,so,audit,aa
```

The Trusted Shell has two states: *ordinary* and *administrative*. Entries with "tsh" in the menus field appear in the menu of the ordinary state of the Trusted Shell, whereas entries with privileged group *g* (e.g., ssa, so, audit, aa) in the menus field appear in the menu of the *g*-administrative state of the Trusted Shell. When the command type is a pathname (p), the Trusted Shell executes the absolute pathname when a user selects it. When the command type is a built-in (b), the Trusted Shell internally executes the command when a user selects it. The built-in commands are: ? (help), *cd*, and *tsh* (a transition from an administrative state to the ordinary state). We can think of the predefined control characters like ^d (^s, ^q, ^h, ^u, ^z^z [the SAK], ...) as built-in commands and enter any of them in *s_cmd* as such, but these are not built-in commands of *tsh*. For a built-in command, the pathname field is ignored by *tsh* and can be empty. When the command type is a state transition (s), the Trusted Shell includes it as a menu item if the user has execute access to the pathname. (A design alternative, currently not implemented, for a state transition menu item to appear is to check group membership.) The *tsh* does not check that the value of the command field for a state transition is a valid group nor a valid privileged group (both TSP and SSA responsibilities).

The *tsh* deals with only three environment variables: TERM, HOME, and PRIVSTATE. The *tsh* reads environment variable TERM for the clear-screen escape sequence, which it plays before printing a help menu. The *tsh* reads environment variable HOME to execute *cd* when you call *cd* with no argument. Also, *tsh* writes environment variable PRIVSTATE to differentiate ordinary from administrative *tsh* states for guard function *privstate()*, which reads environment variable PRIVSTATE to decide after it already knows from /etc/utmp that you must be in the Trusted Shell.

The TSP can customize *tsh* by editing *s_cmd* and by adding or deleting or merging groups and by adjusting DAC on files as necessary. For example, if the TSP introduces a downgrader role with a privileged group, say named *grader*, then the TSP can integrate this role into *s_cmd* with editing, not by remaking and reinstalling *tsh*.

5.3. Command Table Population

It is important to point out that each pathname command in *s_cmd* is a trusted process, even if it has no privileges, because it has access to the trusted path, and therefore is part of the TCB; the TSP should not enter pathnames in the Command Table that correspond to untrusted processes. The security check command of Secure Xenix, *scheck*, verifies that each pathname command has both an absolute pathname and an entry in Installation Table /etc/security/s_install.

6. Experience

Converting the kernel and trusted processes to the new privilege regime took about four persons less than one month. Converting the kernel took one person less than a week. One interesting problem that arose involved the *mkdir* command and the *access()* system call, which we solved by introducing *ipriv*. Another interesting item was the interaction between *setuid()* and privilege vectors. Identifying privileges of a trusted process is simple. For example, *login* is no longer a set-UID root program; it runs with owner bin and privileges MAC EXEMPT, DAC EXEMPT, AUDIT, AUDITLOG, CHOWN, LINK_DIR, MKNO_D, NICE, SETFLABEL, SETPLABEL, SETUID, and ULIMIT.

We have centralized all security relevant installation knowledge in the Installation Table, where each entry has the form

key:pathname:owner:group:mode:acl:label:privileges:links:comment

Currently, the Installation Table contains about 30 privileged programs. Most privileged programs have only a few privileges. As examples, the security check command, *scheck*, has privileges MAC_EXEMPT and DAC_EXEMPT; the *mkdir* command has privileges MAC_EXEMPT, LINK_DIR, MKNOI, and SETFLABEL. Programs *init* and *inir* have all privileges.

Command *scheck* verifies a particular set of security assertions. It checks the integrity of the security tables in directory */etc/security*, it checks the installation of each entry in *s_install*, it checks the *hierarchy assertion* (successive pathname components have nondecreasing security labels), it looks for programs with privileges not recorded in *s_install*, and it looks for other "security lint" like set-UID and set-GID programs. In practice, this command, like *fsck*, is quite useful. While there is no superuser on Secure Xenix, for example, it has been tempting for some developers with role TSP to create and install an omnipotent shell, by making a private copy of *sh* or *csh* and endowing it with all privileges when working as a TSP. Command *scheck* helps find such bad things.

Even with the new privilege regime, it is still important to retain and use the set-UID and set-GID mechanism and to allow set-UID root programs. Set-UID root programs do not need the DAC_EXEMPT privilege to have DAC access to root-owned files. DAC is the solution of first resort; the DAC_EXEMPT privilege is the solution of last resort. Currently, Secure Xenix has only one set-UID root program and it has no privileges, so it just changes DAC context.

Developing the table-driven trusted shell framework for both ordinary and administrative users also took less than one month, once we decided to make it table-driven. Since it is small, simple, and nonprivileged, the trusted shell interpreter has been very stable; we now focus on the contents of the Command Table. For example, changing the role that can kill rogue processes from SSA to SO is now trivial.

We plan to write a separate paper that describes more of our experiences using the privilege regime described in this paper.

7. Conclusions

We conclude from this experience that you can enforce the LPP from a UNIX-based operating system and maintain compatibility of nonprivileged programs, and that such a transition is doable (minor, not major, surgery on the kernel and privileged programs), desirable for security, and useful. If you partition the superuser role and privilege, then it decreases administrative procedures that you would need otherwise. Also, such partitioning decreases the assurance effort; divide-and-conquer simplifies both testing and reasoning about privilege because there is less potential interference among privileged processes. Furthermore, emerging UNIX standards like POSIX should not preclude non-superuser-based privilege regimes because doing so may preclude tighter security.

Acknowledgments

John Woodward of MITRE Corp. provided us with some hints on the general notion of a granular privilege mechanism. Doug Steves of IBM ISP, Ken Witte of IBM ISP, and Jeremy Liang of IBM FSD provided helpful comments on the ideas in this paper. We thank Gerry Ebker and Walt Harner of IBM FSD for support and encouragement.

References

- [B] Bishop, M., "Managing Superuser Privileges under UNIX," Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California 94035 (June 1986).
- [G+] Gligor, V.D., *et al.*, "Design and Implementation of Secure Xenix," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 2, pp. 208-221 (February 1987).
- [G] Gligor, V.D., "Guidelines for Trusted Facility Management and Audit," unpublished report, available from the DoD Computer Security Center (December 1985).
- [K] Kramer, S., "LINUS IV - An Experiment in Computer Security," *Proc. 1984 Symposium on Security and Privacy*, Oakland, California, pp. 24-33 (April 1984).
- [POSIX] *Portable Operating System for Computer Environments*, sponsor Technical Committee on Operating Systems of the IEEE Computer Society, P1003.1, Draft 9, unapproved draft (January 2, 1987).
- [S] Saltzer, J.H., "The Protection and Control of Information Sharing in Multics," *Communications of the ACM*, Vol. 17, No. 7, pp. 388-402 (July 1974).
- [SS] Saltzer, J.H., and Schroeder, M., "The Protection and Control of Information Sharing in Computer Systems," *Proc. of IEEE*, Vol. 63, No. 9 (September 1975).
- [TCSEC] Department of Defense Computer Security Center, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD (December 1985), which supersedes CSC-STD-001-83, Library No. S225,711 dated August 15, 1983.

	acct()	- enable or disable process accounting
*	audit()	- enable or disable auditing
*	auditlog()	- append record to the audit log
	chroot()	- change the root directory
	lock()	- lock a process in primary memory
+	mknod()	- make a directory or a special or ordinary file
	mount()	- mount a file system
	plock()	- lock process, text, or data on memory
*	setfbl()	- set file label
*	setfpriv()	- set file privileges
*	setfsbl()	- set file system labels
*	setplbl()	- set process label
*	setuname()	- set node name of current system
	shutdn()	- flush block I/O and halt the CPU
	stime()	- set the time
	umount()	- unmount a file system
*	vhangup()	- virtually "hangup" the current control terminal

Notes

- * new security-specific system call
- + for files types other than named pipe special

Figure 1. Secure Xenix Privileged System Calls.

@	aclcreat(), aclopen(), aclquery(), aclrm()	- ACL operations
@	chmod()	- change mode of a file
@	chown()	- change the owner and group of a file
;	fork()	- create a new process
@	kill()	- send a signal to a process or a group of processes
#	link()	- link a new file to an existing file
\$	msgctl()	- provide message control operations
&	nice()	- change priority of a process
=	procttl()	- control active processes or process groups
\$	semctl()	- control semaphore operations
!	setgid()	- set group ID
!	setuid()	- set user ID
\$	shmctl()	- control shared memory operations
	signal()	- specify what to do on receipt of a signal
:	ulimit()	- get and set user limits
%	unlink()	- remove directory entry
@	utime()	- set file access and modification times

Notes

- @ owner or superuser
- ; to exceed limits
- # The superuser can link a directory.
- \$ The superuser can use commands IPC_RMID and IPC_SET.
- & The superuser can specify a negative incr value.
- = The superuser can send a command to all processes (except 0 and 1).
- ! caller is superuser, or argument is either the real or effective ID
- | to catch a SAK (Secure Attention Key) signal
- : Only the superuser can increase a process's file size limit.
- % The superuser can unlink a directory.

Figure 2. Secure Xenix Nonprivileged System Calls with Privileged Options

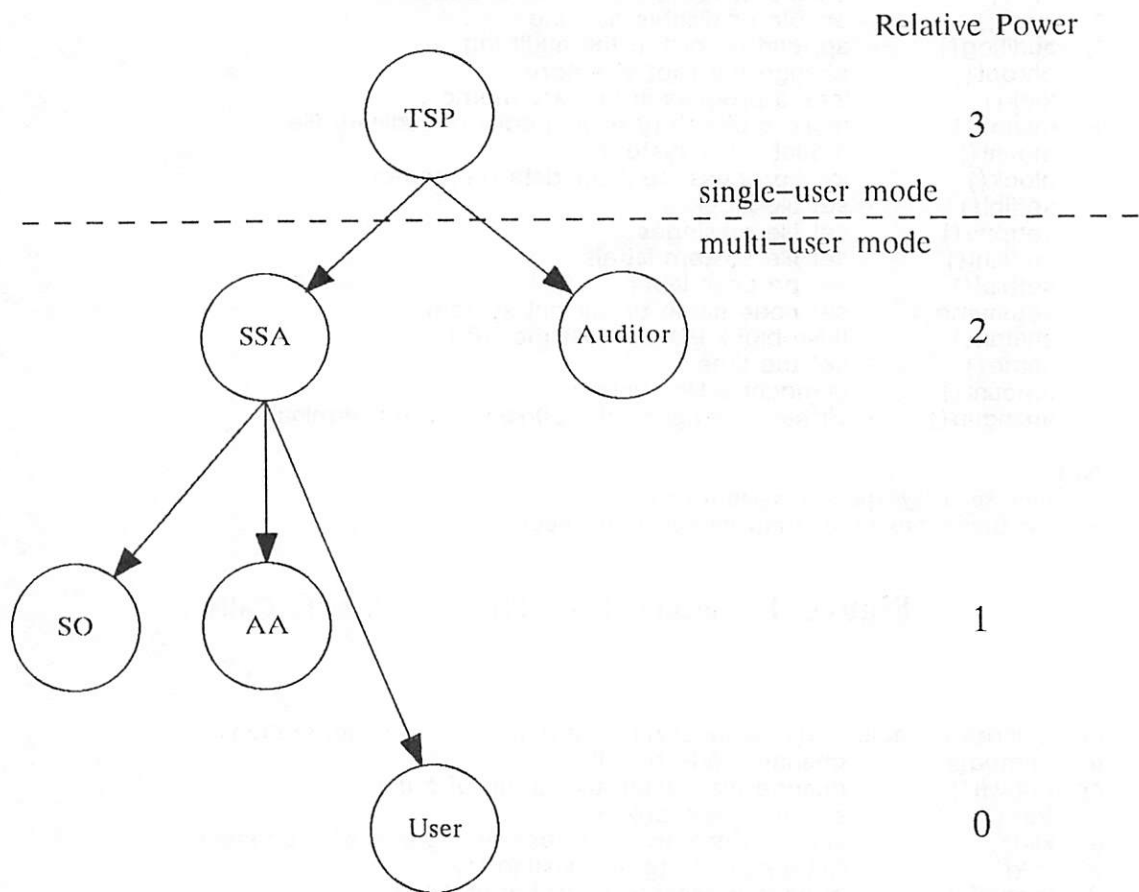


Figure 3. Power Hierarchy of Secure Xenix Roles.

A PARTIAL MODEL FOR A B-LEVEL UNIX®

Frank Knowles
Gould Computer Systems Division
1101 E. University
Urbana, IL 61801
217-384-8599
knowles@gswd-vms.arpa
ihnp4!uiucdcs!ccvaxa!knowles

Abstract

This paper discusses, in the context of a partial security model, the principle aspects of an integrated integrity and label policy for a B1-level secure Unix operating system. The integrity mechanism is the login environment as implemented by the standard chroot system call. A re-working of directories -- interpreting them as corridors rather than containers and using directories of different types -- implements the label policy as mandated by the Orange Book for a Unix file system. Several features of Gould's C2-rated UTX/32S system are also described.

1 INTRODUCTION

In designing a secure Unix® system, two problems of design stand out: ensuring system integrity and creating flexible rules for accessing files with security labels. In this paper, in the context of an informal and partial system model, we present one solution to these problems.

The model we present is much influenced by earlier work, particularly that of Bell and La Padula [1], and Kramer [5]. In approach and scope, the model follows that of the Bell - La Padula Model. Some differences are: the integration of integrity and security policies, and different directory access rules. The notion of directory types, and one of the types, is taken from Kramer's Linus IV.

The concept of a login environment is both an old idea and a new idea. Placing a process in an environment models the standard Unix **chroot** system call. Recall that, in Unix, each process has its own notion of the file system root. This makes it possible at login time to limit the file name space of a login process (and its children) to a proper subtree of the file system. Thus, a *login environment* is a distinguished subtree of the file system in which processes can be placed at login time. In this sense, environments are hardly new. However, as a mechanism systematically used to provide system integrity, environments ARE new, and were first implemented in Gould's C2-rated system, UTX/32S (Miller [6]). This is somewhat surprising since login environments have long been used for special purposes such as boxing in questionable accounts (Wood [7]).

Retrofitting security on a hierarchical file system is an opportunity to impose awkward methods of file access. To preserve, as much as possible, familiar modes of access, this model presents three directory types, one of which is intended to eliminate an undesirable feature of the Multics-style directory access rules (page 107, Gligor [3]). A key idea is a different view of directories, previously described in Knowles [4]. The container metaphor is the usual way of thinking about directories -- certainly it is re-enforced by common speech. That is, a directory "contains" files or, at least, information about files. Thus, a directory can be "read" to obtain the names of the files "in" the directory. We offer another view. A directory is a corridor. Files are offices off the corridor. You must go through the corridor to get to the offices, but access to the corridor does not guarantee access to a particular office (it may have a door of its own).

In this view, a directory and a file entry "in" the directory are separate objects with separate labels, the label of the file entry being the label of the associated file. The corridor metaphor as a way of looking at directories is such a simple idea that one hesitates to say that it could ever have been original. Nevertheless, the access rules inspired by the metaphor have some advantages over the Multics-like rules in the Bell - La Padula Model or those derived from them.

In order to focus on these mechanisms we present a partial model for a secure Unix system. We are concerned with mandatory access rules (rules about security labels) and not with discretionary access rules. We deal only with the file system and basic file system access -- read, write, and execute access modes -- and with sockets (just so we can model communication between environments).

This simplification comes at a cost since, strictly speaking, the merits of a submodel can't be decided in isolation from the rest of the model. However, this model is based upon an on-going B-level Unix design effort at Gould Computer Systems Division, so the design choices presented here were actually arrived at in a wider context.

Nothing in this paper commits Gould Computer Systems in any way whatsoever.

2 THE REQUIREMENT

The design specification and design documentation B1 requirements, as given in the Orange Book [2] are:

3.1.3.2.2 Design Specification and Verification

An informal or formal model of the security policy supported by the TCB shall be maintained over the life cycle of the ADP system and demonstrated to be consistent with its axioms.

3.1.4.4 Design Documentation

Documentation shall be available that provides a description of the manufacturer's philosophy of protection and an explanation of how this philosophy is translated into the TCB. If the TCB is composed of distinct modules, the interfaces between these modules shall be described.

An informal or formal description of the security policy model enforced by the TCB shall be available and an explanation provided to show that it is sufficient to enforce the security policy. The specific TCB protection mechanisms shall be identified and an explanation given to show that they satisfy the model.

Bolded text (in the original) is text applicable to a B1 level system but not to a C2 level system.

3 STATE MACHINE MODELS

This model is a state machine model. A state machine model consists of

states	the elements of the model whose values define current status
events	operations that change state elements thereby causing a transition from one state to another

A security state machine model also includes

state invariant	a relationship between state elements that defines a secure state
event constraint	either a single constraint on all events or the collection of constraints on individual events which help in showing that each event acting on a secure state produces another secure state
security policy	a single name for the state invariant and event constraint(s)
proof of consistency	an argument showing that there are no contradictory axioms in the model
proof of security	a specification of an initial secure state and an argument that each event maps a secure state into another secure state, thus showing that all states of the model are secure states

In this paper we ignore nearly all of this baggage, and concentrate on describing the one or two features that we wish to emphasize.

Security models come in several flavors. The model described in this paper is an access control model patterned after the Bell - La Padula model. That is, the

security policy is concerned with controlling access of subjects to objects and the transfer of information between objects. Covert channels are not addressed.

4 DEFINITION OF A STATE

A "state" is the collection of the values of all the elements of the model. More formally, a *state* is a set of all tuples, (*element*, *value*).

State elements and supporting definitions can be divided into these categories:

Subjects	Elements that use events to manipulate objects.
Objects	Elements that are affected by events. Access modes are defined for each object.

4.1 Subjects

Subjects are the active elements of the model; they request access to objects. Subjects are processes and, indirectly, the users on whose behalf the processes act.

4.1.1 Users

A *user* is a human who interacts with the system. A user gains access to the system via the **login** event. A successful **login** event will result in the creation of an interactive login process which inherits its security characteristics from the security characteristics of the user. From that point on, access to the system by the user is via the login process and its children.

There are several kinds of users:

1. ordinary users
2. administrators
3. ordinary pseudo-users
4. the pseudo-user, ROOT

An administrator is a user that is authorized to log into the System Environment, and an ordinary user is one that is not authorized to do so. The System Environment is used for system administration (including security administration) and is naturally restricted to trusted individuals.

A pseudo-user is a user that does not represent any human user. Pseudo-users are useful as owners of files protected by access control lists. One particular pseudo-user, ROOT, is special because its id (as log id or user id) may confer certain privileges on a process. ROOT is not recognized by the **login** event as a user id, thus no login process can have ROOT as its log id or user id. Only the Trusted Computing Base (TCB) can create processes with an id equal to ROOT.

Each user (including ROOT) has a maximal label which may be assigned to any process with that user id.

4.1.1.1 Security Characteristics

Here is summary of user security characteristics:

user id	unique identifier attached to each user and inherited by a login process
login environment set	set of ids for environments a user may log into
maximal label	an upper bound for labels of processes with the user's user id

4.1.2 Processes

Processes are the means by which users interact with the system. The **login** event creates a login process for the user. That process and its children are the means by which the user interacts with the system.

A process has a *log id* which is set to the user id of the user on whose behalf the process is acting. The purpose of the log id is to facilitate the audit of user activity. Only system processes can change their log id and they do so only as subprocesses created to handle a user requests.

Each process has a user id which, in the case of login processes, is the same as the log id. The *user id* indicates the current state of privilege in which the process is acting or indicates a temporary change of user identity. For instance, when a system process forks a subprocess to handle a user request, the subprocess changes its log id to that of the user (thus becoming a super-user process as distinguished from a system process), and, depending on the task at hand, may retain its ROOT user id if its needs a ROOT privilege or it may change its user id to that of the requestor (thus becoming a user process as distinguished from a super-user process).

A process also has an environment id. Environments are described in a different section, but it suffices to say that the environment id of a process indicates the extent of the file name space of the process, and it is assigned at login time and inherited by subprocesses. System processes, unless they explicitly set their environment id to something else, work in the System Environment (the entire file space).

All processes have security labels. Only a system process or a super-user process may change its label.

4.1.2.1 Security Characteristics

Here is a summary of the security characteristics of processes as discussed in the previous section:

log id	identifies the user on whose behalf this process is acting, or the pseudo-user, ROOT
user id	equal to the log id or another id which is temporarily used

environment id identifies the environment of the process
security label the label at which the process is functioning

4.1.2.2 Processes and Privilege

There are three kinds of processes, listed in order of their privilege, system processes being most privileged:

system process created by the TCB for its own purposes
super-user process created by the TCB to carry out a user request
user process a login process or descendant of such a process

Each process type has all the privileges of an inferior type plus some additional privileges. Any process with environment id set to that of the System Environment and user id set to ROOT is a *root process*. Both system and super-user processes are root processes.

4.1.2.3 System Processes

System processes are created at system initialization and have log id and user id set to ROOT, and environment id set to that of the System Environment. The security label of a system process depends on the purpose of the process. A system process has some privileges that a super-user process does not have. For instance, a system process may change its log id -- something a super-user process cannot do.

4.1.2.4 Super-User Processes

A *super-user process* is a process created by a system process to satisfy a user request and differs from a system process in that its log id is not ROOT.

A typical example of this situation is a system process acting as a trusted server. It forks a process to handle a specific request. That process changes its log id to that of the user, thus becoming a super-user process. The super-user process may change its label and user id depending on the task it is performing.

4.1.2.5 User Processes

A *user process* is a process that is not a super-user process or a system process.

All login processes are user processes, where a *login process* is that process created when a user successfully invokes the **login** event. A login process has both its log id and its user id set to the id of the user. The environment id is set to the environment being logged into. The label of a process depends on the label requested.

4.1.2.6 Inheritance of Privileges

When a process creates another process, the child process inherits all of the security characteristics of the parent. A root process may change some of its security characteristics, but no process may change the security characteristics of another process.

Changing the environment id to that of an environment different from the System Environment also forfeits privileges. During the the **login** event, a system process does exactly this to create a login process. Precisely what privileges should be available to processes executing within a restricted environment is a topic for discussion.

4.2 Objects

Objects are those elements of a state that can be modified by by events. Subjects can also be objects since, among other things, they can be created and destroyed.

Access to an object is controlled by environment checks, and label checks. All subjects and objects have an environment and a label.

4.2.1 Labels

Labels are used to indicate the sensitivity of an object or the trustworthiness of a subject. By a *security label* or just *label*, we mean the structure as described in the Orange Book [2]. Nothing in this model depends on the internal structure of a label (e.g., levels and compartments) so we won't describe it, however, since the order properties of labels are needed in order to reason about sequences of events, we include a definition of the "dominates" relation.

In what follows, A and B are arbitrary labels. We define a relation on labels, *dominates*, that has the following properties. First, for any two labels, A and B, at least one of the following is true:

1. A dominates B
2. B dominates A
3. neither of the above is true, so A and B are *not comparable*

Second, the following axioms hold:

1. A dominates itself.
2. If A dominates B, and B dominates A, then A is the same label as B.
3. If A dominates B, and B dominates C, then A also dominates C.

4.2.1.1 Mandatory Access Control

By *Mandatory Access Control* we refer to those checks involving labels that are made in order to validate an access request. If a Mandatory Access Control (MAC) check fails, the request is denied. Note that an access cannot take place unless the Environment checks and the Mandatory Access Control checks are successful.

4.2.2 File System Hierarchy

The file system is a single directed tree with directories as the non-leaf nodes and data files as the leaf nodes.

For simplicity, we don't model links. Links should be implemented so that access using links is equivalent to some (less convenient) access without them. All files have precisely one parent directory.

Two distinct directories determine subtrees such that one is a proper subtree of the other or the two subtrees are disjoint. A directory is the "root directory" for the subtree under it. There is precisely one *root* directory whose parent directory is itself.

4.2.3 Login Environments

Environments are used to group files and processes into functional groups (which may cut across label boundaries). A principle use of environments is to separate administrative activity from all other activity.

There is a list of file system subtrees known to the TCB as *login environments* (or just *environments*). An environment is characterized by its root directory.

The **login** event associates each login process with an environment. The environment of a process determines the file name space of the process. This association is implemented using the Unix system call, **chroot** (1). Thus a process may directly access only those files beneath the root directory associated with the environment id of the process.

4.2.3.1 System Environment

The *System Environment* is the complete file system. The System Environment, as a set of files, includes all other environments. This is the environment for administrative action. Those files necessary for system maintenance or security administration are kept in that part of the System Environment that lies outside of all restricted environments.

4.2.3.2 Restricted Environments

A *restricted environment* (RE) is an environment whose root directory is a directory different from the root directory of the System Environment. Thus, the file system within a restricted environment is a proper subset, as a set of files, of the complete file system. Gould's UTX/32S, Version 1.0 has just one restricted environment, for all non-privileged users, but the model makes no restriction on how many REs there are.

4.2.3.3 Some Integrity Rules for Environments

Environments as sets of files form a partial order. Direct access is inherently denied going upward or going sideways -- the file isn't in the name space of the process! Going downward, as from the System Environment to a Restricted Environment, certain restrictions are advisable:

- only a root process may write a file lying in an environment whose id is different from that of the process' environment id
- no process may execute a file lying in an environment whose id is different from that of the process' environment id -- confine Trojan Horses!

4.2.4 Directories

Directories are a means of specifying the path to a file. They are corridors that allow access to a subtree of files. Associated with each directory is a set of File Info Blocks. File Info Blocks (FIBs) are described in the next subsection, but, for now, just think of them as the entries in a directory with the difference that a FIB has a label of its own which may be different from that of the directory.

To read a directory means to (attempt to) read the File Info Blocks associated with the directory. Suppose that subject S is granted read access to a directory D which has File Info Blocks, F1, F2, and F3 associated with it. It may turn out, after comparing the security label of S with the label of each FIB, that S may read only F1 and F3. So be it -- reading that directory yields the information in F1 and F3, and as far as S is concerned the files associated with F1 and F3 are the only files in the tree immediately below D.

4.2.5 File Info Blocks

A *File Info Block* is an object that contains the name of an *associated file*. Every file has its FIB. The FIB is created when the file associated with it is created, inherits the label of the associated file, and is destroyed when the file is destroyed. The FIB connects its associated file with the its parent directory. This connection is implemented by reserving space for FIBs at known offsets in the directory inode.

4.2.5.1 Justification of File Info Blocks

Certain obvious high bandwidth covert channels that use file name entries in a directory are eliminated by putting file names into FIBs where they are protected by the label of the file itself. Other smaller bandwidth channels still exist since all processes share the same name space beneath the directory -- if the directory is a non-partitioned directory.

Multics-style label rules require that to write a directory, a process must have a label equal to that of the directory. If a newly created file has a label higher than that of the directory, the creating process cannot immediately edit or even read the file it has just created. Some intervening action such as a **logout** and a **login** in the case of Secure Xenix (page 107, Gligor [3]) or a change of process label in the case of the Bell - La Padula Model (page 20, Knowles [4]) must take place. With FIBs, the process may be at the level of the new file and create the FIB and the file

without writing the directory.

4.2.6 Data Files

A *data file* is a file that contains user data, as distinguished from a directory which contains system data. For simplicity, we let data files include executable files.

4.2.7 Some Label Rules for Files

These rules are for access to directories, data files, and File Info Blocks which we refer to collectively as "files" in this section. The rules are consistent with the Mandatory Access Control requirements as stated in the Orange Book [2].

- A process may read or execute a file only if the label of the process dominates the label of the file.
- A process may write a file only if the label of the process is the same as the label of the file.

4.2.8 Directory Types

It is anticipated that directories will be used in three fundamentally different ways in a typical secure Unix system. Each of these ways is a response to the interaction of label control and directory access.

4.2.8.1 Single-Label Directories

The simplest way to distribute labeled files within the hierarchy is to let directories partition labels as well as files. That is, all files beneath a directory have the same label as the directory. When this is true, Mandatory Access Control is invisible, and users will not have the problem of seeing only a some portion of their files in a directory in any particular login session. So, a *single-label directory* is a directory for which the TCB enforces the following rules:

1. All files beneath the directory have the same label as the directory.
2. All directories beneath the directory are also single-label directories.

No process, not even a root process, may create a file beneath a single-level directory that has a label different from that of the directory.

4.2.8.2 Partitioned Directories

Unix has a directory, `/tmp`, that is used differently than other directories. By convention, utilities create temporary files, as needed, in `/tmp`. So, processes operating with various labels access the same directory at the same time and in all modes: read, write, and execute. The usual label access rules will not allow this situation.

The solution to this problem that we adopt is due to Kramer [5]. The TCB recognizes a partitioned directory, a directory type used solely for the `/tmp`

directory. Before defining a partitioned directory, we need the concept of a *hidden directory for a label*. For any given label, the TCB has the capacity to create within a partitioned directory a subdirectory (invisible except to root processes) which is a single-level directory for that label. A hidden directory cannot be created or destroyed or explicitly accessed except by root processes. Whenever a file is created within a partitioned directory, that file is put into the hidden subdirectory for the label of the file. The hidden directory is created if it doesn't already exist. All files under a partitioned directory are partitioned among hidden directories, and each hidden directory is a single-label directory.

A user process accessing a partitioned directory will access the files in the hidden directory for the label of the process. Root processes may access hidden directories as if they were ordinary single-level directories. To make this all work right, the FIBs in a partitioned directory are specially named and marked as associated with hidden directories.

A *partitioned directory*, then, is a directory for which the TCB enforces the following rules:

1. Whenever a file is created in the directory, it is put into a hidden directory associated with the label of the file.
2. There is only one hidden directory for each label; it is created as needed; and it is a single-label directory.
3. Only a root process may create or destroy or directly access a hidden directory.
4. User processes, when accessing the partitioned directory access files under the hidden directory associated with the label of the process.

4.2.8.3 Non-Partitioned Directories

It is desirable to have a way to build a hierarchy of files at different labels, and, at the same time, avoid partitioned directories because they are difficult to administer. Directories that are not single-level directories or partitioned directories are *non-partitioned directories* or "normal" directories.

In non-partitioned directories, the File Info Blocks are the principle access barrier between processes with one label and files at a higher label.

4.2.9 Root-owned Files

An added protection for important files in Gould's UTX/32S is the protection provided by the kernel to files whose owner is ROOT. In terms of the model we say that any file whose owner is the pseudo-user ROOT cannot be modified except by a root process. In UTX/32S, this provides an added measure of protection for system binaries in the restricted environment where users cannot obtain root processes.

4.2.10 Security Characteristics of Files

This is a summary listing of the security characteristics of Directories and Data Files:

1. parent directory id
2. environment id
3. label
4. file owner
5. directory type
6. if a directory, list of associated FIBs
7. if a data file, associated FIB
8. if a directory, an environment flag

The environment flag indicates if the directory is an environment root directory. If so, it is required that the file owner be ROOT.

This is a summary listing of the security characteristics of File Info Blocks:

1. associated directory
2. associated file (may be a directory also)
3. label
4. hidden-directory flag

4.2.11 InterProcess Channels

Though the details differ, each interprocess channel is an alternative to the file system, and as such must be constrained in the same fashion as is file access by labels, and environments. The inclusion of messages as model objects indicates how interprocess channels in general should be controlled.

Secure Sockets are an exceptional case since they are the chosen vehicle (in UTX/32S, anyway) for inter-environment communication services. Clearly Secure Sockets in their role as trusted servers must also be a part of the model.

4.2.11.1 Messages

Messages are data buffers arranged in queues. It is the queues that are created and destroyed, and the individual messages that are sent (queued), and received (taken off the queue -- destroyed).

When a process creates a message queue, the label and the environment id of the process are assigned to the queue. They cannot be changed. Any process, except a root process, that wishes to send or receive a message through a queue must have the same environment id and label as that of the queue.

It is important that user processes not be able to use messages to bypass label or environment restrictions on data flow.

4.2.11.2 Secure Sockets

A socket is a communication endpoint designed to be the connecting link between two or more communicating processes. Secure sockets are used to implement trusted servers, and that is the only aspect of Secure Sockets that we discuss.

Typically, the server, a root process, creates the server secure socket and binds it to a known address and listens for connections. A client (usually in a restricted environment) creates a client socket and connects it to the server socket. The two processes exchange data until the server or the client closes their socket.

Since the server side controls the exchange of data, the label of the socket is not an issue. (The chief virtue of Gould's Secure Socket mechanism is that the kernel provides non-spoofable client identification information to the server at connection time.)

It is important that only secure sockets are used -- and then only under the control of a root process -- to transfer data between environments.

4.2.12 Devices

In a Unix operating system, access to a device is treated as access to a special type of file named for the device. Since it is desirable to exercise more control over device access than that over ordinary data files, Gould's UTX/32S uses kernel-resident device ownership tables to allocate devices. This model reflects the essentials of that mechanism.

There are three types of devices:

privileged devices	be accessed directly only by root processes
allocatable devices	allocated and deallocated by root processes -- only the current owner (or a root process) can access a device that is not free
other devices	access is controlled by protection bits on the device special file

An example of a privileged device is the root disk partition. An example of an allocatable device is a tty. An example of an "other device" is a pty.

LABEL CHECKS

Devices have two static labels (a maximum label and a minimum label) and a dynamic label (the working label). The maximum label dominates the minimum label, and the working label must always be in between. Whenever a process gains access to a device, the TCB assigns the label of the process as the *working label* of the device.

INTEGRITY CHECKS

Privileged device files are outside all restricted environments. Direct access to devices within a restricted environment is limited by the device special files placed within that restricted environment.

OWNERSHIP CHECKS

Access to an allocatable device is via a trusted server which assigns the user id client as the owner of the device, a fact recorded in a kernel-resident table. All access to the device is restricted by the kernel to processes whose user id matches the owner id of the device (protection bits are irrelevant). Root process may override this check.

4.2.13 The Trusted Computing Base

We can summarize our description of system elements by defining an important design concept for secure systems, the Trusted Computing Base (TCB) as it appears in this model.

The *Trusted Computing Base* consists of:

1. the files residing outside of all restricted environments
2. all root processes and all processes created by root processes except for those processes placed into a restricted environment
3. all administrative users

This implies that the TCB does not extend to files residing in a restricted environment or to processes running within a restricted environment (that is, the environment id of the process is that of a restricted environment). Note that some processes created by the TCB may be user processes, created to handle a specific user request. Such processes are not root processes, but are necessarily *trusted processes*. This use of the terms **TCB** and **trusted** are intended to conform to the usage in the Orange Book [2].

4.2.14 Parting Thought

The model presented in this paper, as far as it goes, is adequate for a B2 level Unix, but there are requirements at the B2 level which we haven't discussed:

1. secure path
2. covert channels
3. least privilege
4. kernel modularity

Of these, the last may prove to be the bugbear.

References

1. Bell, D. E., La Padula, L. J., "Secure Computer System: Unified Exposition and Multics Interpretation," Mitre Corp., MTR-2998, (available as NTIS AD-A023588) (1976)
2. "Department of Defense Trusted Computer System Evaluation Criteria" DoD 5200.28-STD, National Computer Security Center, Fort Meade MD, (Dec 1985).
3. Gligor V. C., et. al., "On the Design and the Implementation of Secure Xenix Workstations," Proceedings of the 1986 Symposium on Security and Privacy, IEEE Computer Society (Apr 1986).
4. Knowles, Frank, "A Discussion of Formal Models," a paper presented at the Special Interest Group on Formal Methods of the British Computer Society at their annual meeting in London, (Dec 1985).
5. Kramer S., "Linus IV -- An Experiment in Computer Security," Proceedings of the 1984 Symposium on Security and Privacy," IEEE Computer Society (Apr 1984).
6. Miller, Greta, et. al. "Integrity Mechanisms in a Secure Unix: Gould UTX/32S," AAIA/ASIS/DODC 2nd Aerospace Computer Security Conference, A Collection of Technical Papers (Dec 1986).
7. Wood, Patrick H., Kochan, Stephen G., "Unix System Security," Hayden Book Co., Hasbrouck Heights, N. J. (1985).

A Remote-File Cache for RFS

M. J. Bach*
M. W. Luppi**
A. S. Melamed
AT&T Bell Laboratories
Holmdel, New Jersey
K. Yueh
AT&T Information Systems
Summit, New Jersey

ABSTRACT

Remote File Sharing (RFS) in UNIX*** System V caches remote-file data on client machines. While caching improves the system performance, it raises the issues of preserving consistency among multiple copies of file data. The design we chose ensures that cached data is identical to the remote file content at the time of user access from any machine on the network. The paper describes the architecture of the client cache and the measured performance gains. It considers a number of alternative schemes for preserving consistency and provides the rationale for the scheme chosen.

1. INTRODUCTION

Remote File Sharing (RFS) in UNIX System V Release 3.0 allows transparent file access across machine boundaries over a network [5]. When a process makes a remote access, the local ("client") machine sends a request to a server machine, which continues the operation.

In Release 3.1 RFS, we have implemented caching of remote-file data. The client machine now keeps the data blocks for a remote file in a local memory buffer pool, also called the "client cache". A remote system call may find the requested data in the cache, avoiding a network access. The consequent reduction in remote traffic can considerably improve user response times and remote-access system capacity. This approach raises, however, the classic issues of preserving consistency among multiple distributed caches and the server file image.

When designing client caching we had several objectives in mind:

- Achieving a substantial performance gain for remote access in most workload situations,
- Ensuring transparency of the caching mechanism to RFS users,
- Guaranteeing consistency between client-cached data and the actual contents of the server file,
- Ensuring compatibility with non-caching RFS releases, and
- Minimizing changes to Release 3.0 RFS.

We believe these objectives have been met. In particular, the client cache has resulted in performance gains of up to 200 percent for some user workloads. Our design preserves the local file system semantics and ensures that all clients have the same view of server files. Users can

* Author's current address is IBM Israel Scientific Center, Technion City, Haifa, Israel.

** Author's current address is Morgan Stanley, 1251 Avenue of the Americas, New York, NY 10020

*** UNIX is a Registered Trademark of AT&T.

expect the same relationship between RFS cache buffers and a server file as presently exists between local disk buffers and a local disk-resident file*. There are no "windows" where RFS users can find obsolete data in the cache. In other words, a *read* of cached data always includes the results of all previous modifications to the file image.

In contrast, caching systems enforcing a weaker degree of consistency than this may require that applications be substantially modified in order to work correctly in a distributed environment. This is especially true of applications which employ inter-process synchronization mechanisms (e.g. named pipes) across the network. Race conditions abound in such an environment.

This paper describes the specifications and the architecture of the client-caching in UNIX System V Release 3.1. Subsequent sections give an overview of SVR3.1 client caching features: handling of *read* and *write* data, buffer-pool management, special cases of the general caching mechanism, and maintenance of cache consistency. We also discuss some alternative approaches that were considered during the design phase.

2. THE NETWORK CACHE

The enhanced RFS caches *reads* of remote regular files only. Since internal studies of various environments indicated that *reads* of regular files occur with very high frequency in typical user workloads, we felt that this caching policy should yield substantial benefits while keeping the design simple.

Client-machine caching is activated at *mount* time for all regular files in a mounted remote resource. Users can prevent the activation via a mount option, in which case all *reads* and *writes* of the files in a mounted directory will be uncached. The override capability can be useful for applications which do their own caching of remote files.

The following sections describe how the *reads* and *writes* of the cached data are handled, how the cached data is managed in the local buffer pool, and why some regular files and non-regular files are not being cached.

2.1 Read/Write Handling

Individual remote-file blocks are kept in the local system buffer pool. When a client process reads a server file, the client system searches the buffer pool. Each cache "hit" results in a data-copy from the buffer pool to the client process data area. Each "cache miss" generates a *read* request to the server; the returned data blocks are added to the local buffer pool.

A cache "hit" occurs only when *all* of the requested data is found in the buffer pool. If any data block is missing, the system treats this as a cache "miss" and requests all of the data in a single "transaction" from the server. (This situation can arise for *reads* spanning several buffers.) Requesting the data as a unit provides a simple way of preserving the UNIX file system semantics for standalone-processor access, which guarantee each *read* to be atomic. Requesting only the "missed" data could violate the "atomicity" for remote accesses, since the "hit" blocks found earlier in the cache and the "missed" blocks returned later from the server could reflect two different remote-file states.

Remote *writes* are immediately sent to the server, after having been "written through" the local cache. This write-through policy simplifies both the maintenance of data consistency and system recovery.

* There is an exception to this general policy, concerning block special files.

2.2 Buffer-Pool Management

We adopted a shared cache approach, employing an LRU replacement algorithm. A common pool of buffers is shared between the RFS and local disk traffic. In order to control the adverse effects of competition between the two traffic streams, the buffer pool is partitioned into three areas: 1) buffers reserved for local traffic, 2) buffers reserved for remote traffic, and 3) buffers that "float" between local and remote use. The partitioning is controlled by tunable parameters, which can be set to allow for full sharing or a static division of the buffers between the two activities. (However, if RFS is not running, or if there has been no recent RFS activity, the local traffic has unrestricted access to the entire buffer pool.)

Without the above partitioning it is possible for local traffic to monopolize the entire buffer pool. (That is because of shorter local as compared to remote access times and the LRU buffer replacement algorithm employed.) The above effect is undesirable since the replacement costs of remote data are much higher than the replacement costs of local data.

Two freelists, for local and remote buffers respectively, are maintained to guarantee the efficiency of searches for available buffers. An additional hash table has been added to enable fast access to all buffers belonging to a remote file. Individual buffers are found by hashing a search key that uniquely identifies a block of remote data. The search-key components are:

- Server machine ID,
- Advertised directory ID,
- File ID, and
- Block number.

2.3 Special Cases

Remote reads of mandatory-locked files are not cached because there is no potential for performance improvement. Because locking information resides on the server machine, a *read* of a mandatory-locked file would require a network message exchange even for a cache hit. *Reads* and *writes* of block special files are not cached as they could duplicate existing data in the cache (e.g. data from a regular file residing on the block device defined by the special file).

We did not cache *reads* of remote directories and *a.out*'s, since in each case it was unclear whether the potential benefit justified the additional design complexity. Named pipes are not cached, because pipe data is read only once after it has been written.

3. MAINTAINING CACHE CONSISTENCY

Maintaining data consistency is a generic problem for distributed environments with data replicated at multiple nodes. There are a number of design alternatives that trade the degree of consistency achieved, user transparency and the performance improvements obtained [1][2][3]. One alternative is for each client to synchronize the cached data with the server-file content at the time of file close only. This does not, however, guarantee a consistent view of the data in the presence of concurrent file access from multiple machines. If several client machines hold a file concurrently *open*, and one of them modifies the file, the remaining clients are not aware of the modification. This introduces windows of inconsistency between a client's cached data and the server file image. One possible solution is to permit only "read-only" access to files that are held *open* remotely by multiple machines. Then, of course, remote access is no longer transparent to user applications.

The described inconsistencies could lead to outright breaks in functionality for applications which use process synchronization across the network. The following scenario demonstrates this:

- i. A process on client machine A modifies a server file. No update notification is sent to the other client machines caching the file data.

- ii. The process on machine A sends a message (via a named pipe, for example) to a process on machine B, which has the file *opened*. The message indicates that (1) the modification has taken place and (2) the modified data can now be read.
- iii. When the process on machine B reads the file, it can find in its local cache old data that does not reflect the update. This means that process synchronization is no longer enforced by the named-pipe message.

Clearly, applications can be designed to avoid the described situations. This places, however, a burden on application programmers to recognize and protect against such occurrences. In particular, applications that work correctly in a standalone environment may have to be substantially modified for distributed environments.

To avoid this this class of problems, RFS enforces a "strong" cache consistency. The basic policy is to ensure that *cached data is identical to the server-file image at the time of user access*. ("Server-file image" is defined as the disk-resident contents of the file, as well as any "delayed-write" buffers associated with the file in the server disk cache). This means that a *read* of cached data always reflects the results of all previous modifications (*writes* and truncations) to the file image.

When a process begins writing a file, the file's data present in other machines' caches may be no longer valid. One approach to maintaining "strong" consistency is to send a notification at each *write* to all other machines currently accessing the file, indicating which part of the file is being updated. Each machine invalidates the buffers in its cache affected by the *write*. In a variant of this scheme, the modified data is included in the notification message, in which case each machine copies the data into the affected buffers, instead of invalidating the buffers.

This approach was ruled out because of the potential for performance degradation. It is easy to envision situations where many processes are sharing a file, and one or more processes are continually writing to the file. In this case, the server would have to send notification of every *write* to each affected client machine. The volume of notification messages could cause performance to fall below that of a system without caching.

To minimize the synchronization cost, the approach we adopted does not send messages for every file modification. It distinguishes between two cases: 1) a client machine has the file opened at the time of modification, and 2) a client has closed the file being modified. These cases are discussed in the following sections.

3.1 Consistency for Multiple Concurrent Access

When all processes accessing a server file reside on the same client machine they share the same cache and therefore have the same view of the data. If processes on more than one machine concurrently access a server file, consistency is threatened whenever the file is modified. Because of the "write-through" policy, the writing-machine's cache is still consistent, but the content of other machines' caches may now be out-of-date.

If a client machine has a remote (server) file open, the server machine sends an "invalidate" message at the first file modification by another machine. The invalidate message causes removal of all the file data from the affected client machine's cache, and temporarily disables the client's caching of the modified file. The temporary disabling avoids possible performance degradation due to invalidate message traffic during periods of heavy write activity. (The implicit assumption is that *write* traffic is "bursty;" a first write is likely to be soon followed by others.)

Subsequent accesses from the client then bypass the client-cache until caching of the file is re-enabled. Re-enabling takes place when the writing processes close the file or when a maximum time interval (a tunable parameter) has elapsed since the last modification of the file. This parameter can be set so as to result in few invalidate messages.

The RFS server suspends the first *write* to a file until all machines respond to the server invalidate messages, indicating that their caches have been purged. This policy avoids race

conditions that could threaten consistency if the *write* were allowed to return before the receipt of all client responses. The following scenario (a variant of the scenario in the previous section) describes such a race condition, and further illustrates the difficulty of enforcing process synchronization in caching environments with "weak" cache consistency.

- i. A process on client machine A *writes* the file. The *write* is allowed to return to machine A without waiting for other machines' responses to the cache-invalidation message sent by the server.
- ii. Machine A receives the "return-from-write" message from the server. The writing process sends a message via a named pipe to a second process on machine B, indicating that the file has been updated and can now be read.
- iii. A race exists between the named-pipe message from machine A and the invalidation message from the server machine. If the named-pipe message were received first, the process on machine B could *read* the server file and find data blocks in local buffers that do not reflect the earlier *write*.

3.2 Consistency across Closes

As long as one or more processes on a client machine have a server file *open*, the server sends invalidate messages to the client, as described in the previous section. After the last *close* on a client machine, the server no longer sends such messages, since in our design it is no longer aware of the client's "interest" in the file. We do not purge file buffers at the file close by a client, because internal studies indicated a frequent re-use of buffers across file closes. The contents of file buffers lingering in the client's cache may therefore become obsolete, and in the absence of further action the incorrect data could be read after a subsequent open.

One possible solution is to have the server keep per-client lists for each advertised file, in order to track modifications that could affect the contents of client caches. When a client re-opens a file, the server could determine from the list information whether the client should purge the file buffers. This approach was ruled out because of the complex bookkeeping involved and the possible propagation of many "per-client" lists.

The approach we adopted was to use *version numbers* to identify changes in file state. When a server file is modified, it is "stamped" with a unique version number (kept as a field in the file's inode-table entry). When a client does a "last close" of a server file, the current version number is stored in the headers of the client buffers associated with the file. When the client later re-opens the file, the new version number (included by the server in the "return-from-open" message) is compared against the previously-saved version number. Any change in version numbers indicates that the file's buffers should be purged from the client cache. Otherwise the buffers can be re-used.

The following is a typical scenario:

- i. Assume an advertised file currently has the version number 504. When a process on client machine A does a last *close* of this file, the value 504 is stored in the client buffer headers.
- ii. Subsequently, the file is modified by a process on the server or another client machine, and is assigned a new version number (676, for example). Client A is not notified of the modification, since it does not currently have the file opened.
- iii. A process on client A re-opens the file. The new version number (676) is returned. If there are any file buffers still lingering in the cache, the old number (504) is obtained from one of the buffer headers. Since the old and new version numbers differ, client A removes the file buffers from its cache.

The described scheme avoids the costs of purging client buffers at every "last" file *close* by the client or sending synchronization messages while the file is closed.

3.3 Maintaining Consistency with a State-ful Architecture

While investigating alternatives for maintaining cache consistency, we became aware of the advantages afforded for this purpose by the RFS's "state-ful" architecture. An RFS server always preserves "state" information such as the number and identity of the client machines accessing a file, and duration of *write* activity. This information allows the system to take appropriate action to preserve "strong" data consistency across multiple machines.

We believe that it would be difficult to enforce the same degree of consistency at the operating system level in a "state-less" environment. It seems that environments lacking "state" information must either follow a more conservative strategy such as turning caching off when files are being locked (thereby losing opportunities for performance gains), or else settle for a weaker enforcement of consistency. In the latter case, application programmers with a need for more rigorous consistency may have to resort to complex strategies to ensure that all windows of inconsistency are eliminated.

Since many recent implementations of distributed file systems now include some form of caching [1][2][3], the degree to which consistency can be maintained may become an important consideration in the debate over "state-ful" versus "state-less" architectures.

4. PERFORMANCE ISSUES

Generally, the degree to which caching improves performance depends on: 1) the percentage of remote *reads* in the workload, 2) the client cache hit ratios, and 3) the cost of maintaining the data consistency.

In situations where most of the remote traffic consists of *reads*, client caching has been observed to run as fast as local disk traffic. For less read-intensive workloads, performance gains are less dramatic, but still substantial. We measured a relative response time improvement of approximately 200 percent for typical read-only data-base benchmarks, executing in a single client-server configuration with 1 to 5 client users. The relative improvement increases for multiple-client single-server configurations. This is because caching reduces the cost of remote file access on the server machine*, which is typically a bottleneck in such configurations.

In general, the benefit of RFS caching substantially outweighs the cost of maintaining consistency [4]. In particular, the overhead of maintaining consistency across closes (Section 3.2) is insignificant, requiring at most an inexpensive purge of out-of-date buffers when a file is re-opened.

The costs of maintaining consistency for concurrent access (Section 3.1) can potentially be greater. In practice, however, we have found it to be generally insignificant; a number of factors mitigate the expense of sending invalidate messages to multiple client machines.

1. The expense is incurred only when processes on more than one machine have the same file concurrently *open*. This situation is rare, except for some database environments.
2. Even when this situation is frequent, disabling the cache during periods of *write* activity avoids the need to send invalidation messages for subsequent *writes*, and thus helps achieve at least a "non-cached" level of performance.
3. The cache disabling is done on a per-file basis only. If the server turns off caching for a file, *reads* of other server files are still cached.
4. Caching is not disabled when a single machine is writing to a file. If a client process *writes* a file, caching is turned off for all other client machines that have the file *open*. The writing

* Caching, of course, also reduces the cost of remote file access on the client machine.

machine continues to benefit from caching, unless a second machine begins to write the file.

5. CONCLUSION

Remote File Sharing in UNIX System V Release 3.1 includes caching of remote files on client machines. The design preserves the standalone-processor system semantics and guarantees that cached data is identical to the server-file image at the time of user access from any client machine on the network. Implementation of this "strong" consistency at the operating system level was possible because of the "state-ful" architecture of RFS.

Caching is transparent to user applications, i.e. applications do not need to be modified in order to work correctly in the distributed environment. The design attempts to minimize the number and cost of the invalidate messages required to maintain the data consistency. The resulting performance gains are substantial for most user workloads.

6. ACKNOWLEDGEMENTS

The caching feature includes the ideas and effort of many people in AT&T IS and AT&T BL. Jerry Feder, Art Sabsevitz, Y.T. Wang, and Larry Wehr made a number of suggestions that were incorporated in the final design. Ron Gomes helped identify the problems inherent in directory caching. Paul Lee and Steve Rago independently suggested the "version numbers" scheme that was ultimately used to maintain consistency across closes. Ron Barkeley, C.T. Chen, and Gerry Gordon played a major role in the performance evaluation and testing of this feature.

We'd like to give special thanks to our supervisors---Jerry Feder, Marilyn Partel, Art Sabsevitz, and Y. T. Wang---who believed in the work.

REFERENCES

- [1] Sandberg, Goldberg, Kleiman, Walsh, Lyon. "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings*, Portland, Oregon. June 1985.
- [2] Schroeder, Gifford, and Needham. "A Caching File System for a Programmer's Workstation," *Proceedings of the Tenth Symposium on Operating System Principles*. December, 1985.
- [3] Satyanarayanan, Howard, Nichols, Sidebotham, Spector, West. "The ITC Distributed File System: Principles and Design," *Proceedings of the Tenth Symposium on Operating System Principles*. December, 1985.
- [4] Kevorkian. "System V Interface Definition," *Spring 1985 Issue 1*.
- [5] Rifkin, Forbes, Hamilton, Sabrio, Shah, Yueh. "RFS Architectural Overview," *USENIX Conference Proceedings*, Atlanta, Georgia (June 1986).

RFS in SunOS

Howard Chartock

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, Ca. 94043
sun!howard

ABSTRACT

This paper describes a prototype implementation of AT&T's Remote File Sharing (RFS) within Sun Microsystems' version of the UNIX[†] operating system (SunOS). The Sun implementation provides complete RFS client and server functionality in the kernel, including support for such features as remote device access and remote locking.

A major goal of this work was to implement RFS within the framework of Sun's Virtual Node/Virtual File System (vnode/VFS) architecture. The vnode/VFS abstraction provides an object-oriented interface in SunOS for filesystem objects and their associated operations. The System V, Release 3 implementation of RFS, in contrast, is based on a remote system call model. Because this model has no adequate associated interface in the kernel, its realization involves major RFS-specific changes in kernel code and data structures. In the Sun implementation, RFS was recast in terms of vnode/VFS objects and operations; it then became possible to simply "plug in" RFS to the vnode/VFS interface, without the need for special modifications to the kernel. The major challenges of this implementation occurred in reconciling mismatches between the remote system call abstraction and the vnode/VFS abstraction.

A second major goal was to show that an RFS can be constructed that functions in a heterogeneous machine environment. The Sun implementation represents one of the first demonstrations of this capability, specifically between Sun Workstations® and AT&T 3B2 computers. A number of heterogeneity issues arose and were resolved in the course of the implementation.

1. Introduction

Remote File Sharing¹ (RFS) is a distributed filesystem provided with UNIX System V, Release 3 that allows a network of machines to transparently share files. An application program running on one machine can access a remote file in the same manner as if it were local; the program need not, nor can it easily, distinguish between local and remote files.

To accomplish this transparency, RFS must mesh with existing UNIX file naming semantics. It does this, in part, by allowing a client machine to mount a directory in a remote filesystem in the same manner as is done for local filesystem mounts. Once a remote directory is mounted, its contents become part of the client's unified hierarchical filesystem name space, thus providing transparency to an application.

At the application level, RFS functionality is quite similar to that provided by other remote filesystems including Sun Microsystems' Network File System² (NFS). However, the underlying design and implementation of RFS and NFS are quite different. Whereas NFS is based on a remote *filesystem* protocol, RFS is a remote *system call* protocol. This difference has profound implications on implementation.

[†] UNIX is a registered trademark of AT&T.

A second notable point concerning the differences between RFS and NFS is that, whereas NFS does not maintain server state on behalf of a client, RFS does. The single most important piece of state maintained is the holding of a server inode on behalf of a remote client. Also, because of this stateful quality, RFS requires a recovery mechanism to allow the server to release resources held on behalf of a client when the connection with that client is lost.

This paper describes an implementation of RFS within Sun's version of the UNIX operating system, SunOS. From the start, a major goal of this implementation was to avoid generic changes to SunOS solely for the support of RFS. To achieve this goal, it was decided to implement RFS in terms of SunOS's Virtual Node/Virtual File System (vnode/VFS) architecture.³ This architecture is designed to support the straightforward implementation of different filesystem types within SunOS. Using the vnode/VFS interface, new filesystem types can be written and "plugged in" to the kernel in much the same manner as device drivers. The 4.2 BSD filesystem, NFS, and an MS_DOS filesystem are currently supported by this interface within SunOS. In addition a vnode/VFS interface has been written for a UNIX System V, Release 2 based kernel and used to implement NFS.⁴

Much of this paper is concerned with the implementation of RFS in terms of the vnode/VFS interface. Recasting RFS in this light provided some interesting insights into the respective natures of the system call and filesystem abstractions and how to reconcile the two.

Another important goal of this implementation was to demonstrate that RFS could support file sharing in a heterogeneous environment, specifically between Sun Workstations running a Berkeley based SunOS and AT&T 3B2 computers running UNIX System V, Release 3. Support for heterogeneity requires that a protocol be designed so that data exchanged between different machines be in an agreed-on representation that both understand. To achieve this, RFS makes use of Sun's eXternal Data Representation format (XDR).⁵ Furthermore, a protocol should not require a machine to know anything about the native representation of data of other machines on the network. This paper discusses some unexpected heterogeneity issues that arose and were resolved, involving the RFS protocol and implementation.

The first step in the RFS port was to get the necessary protocol support up and running between Suns and 3B2s. Client functionality was then implemented in the kernel as a filesystem type under the vnode/VFS interface. Following this, server support was implemented using the operations provided by the vnode/VFS layer. Finally, additional RFS features such as recovery, forced unmount, advertise service for remote resources and miscellaneous utilities were implemented.

2. Protocol Support

RFS requires an underlying protocol to communicate with remote machines. The stateful nature of RFS and the need for recovery following the loss of a connection dictate that the protocol be reliable and that it be capable of detecting and informing RFS when a connection with another machine has gone down.

RFS interfaces with the underlying protocol via AT&T's STREAMS framework.⁶ STREAMS is a mechanism and interface definition that allows modules such as protocol modules and drivers to be connected together dynamically in a well-defined way, so that data may be exchanged between them.

In addition, to achieve independence of the underlying protocol, RFS uses AT&T's Transport Interface⁷ (TI) that provides a transport-protocol-independent layer, into which different protocols may be plugged. TI consists of both an in-kernel interface and a user-level interface in the form of a library. RFS makes use of both of these interfaces.

The first phase of the RFS port required the establishment of suitable network support for RFS. The STREAMS framework was ported to SunOS by Bill Shannon and Glenn Skinner. Using the STREAMS interface, the TI interface code and a proprietary AT&T STREAMS/TI-based protocol, NPACK, were ported to SunOS and tested to provide communications with the 3B2s.

3. System Call vs. Filesystem Issues

One of the principal differences between RFS and NFS is in the abstractions used as models for their design and implementation. RFS uses a remote system call abstraction whereas NFS uses a remote filesystem abstraction.

3.1. The Issues in General

The system call abstraction uses systems calls as the interface by which one manipulates filesystem objects. There are a number of important points to note about this abstraction.

- [1] One argument to the system call is always a reference or handle to a file. This reference takes the form of either a pathname or a file descriptor.
- [2] A pathname relies on the uniform filesystem name space presented by UNIX filesystem semantics. The user of such a name need know nothing about the underlying filesystems and mount points that compose the name, in order to access the file it refers to. Thus, resolution of the pathname across and within filesystems is below the level of abstraction presented by the system call interface. Furthermore, the relation between pathnames and files themselves is many-1, i.e., there can be several names for the same file.
- [3] A file descriptor is returned by a system call and represents a reference to a file which is guaranteed by UNIX semantics to exist until the descriptor is closed.

In contrast to the system call abstraction, the filesystem abstraction typically provides a somewhat lower level of operations on files and filesystems. In addition to Sun's vnode/VFS, a number of other filesystem interfaces have been defined for the UNIX kernel.^{8,9} These interfaces all share several properties:

- [1] A set of operations on filesystem objects is provided out of which system calls are built.
- [2] A *lookup* operation is provided, that resolves a name into a file handle. That is, the operation at least verifies that the file exists and may hold the file in existence for the caller. Furthermore, the operation returns a vnode/inode that contains a handle to the file.
- [3] Operations other than the lookup op typically require one or more of these file handles as arguments, specifying the associated file(s).
- [4] The filesystem interfaces do not deal with certain issues such as the mount abstraction.

As the above discussion suggests, an intermediate layer resides below the system call layer and above the filesystem layer. This, *meta-filesystem* layer bears responsibility for pathname evaluation, and deals with issues such as the crossing of mount points, and symbolic link substitution.

The relationships among these abstractions is summarized in Figure 1, which gives an example of how a system call operation such as *open()* is broken down into meta-filesystem and filesystem-specific operations. The arrows represent the fact that in an implementation, routines at one level make calls on routines in the levels below them.

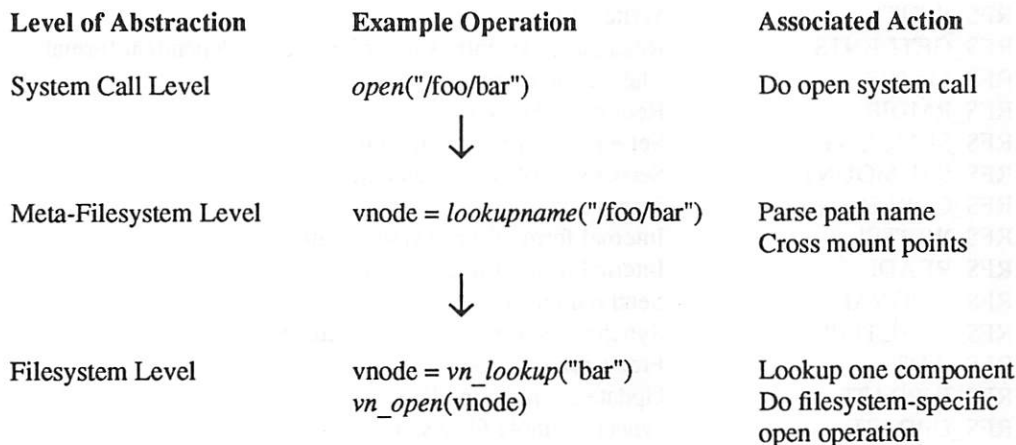


Figure 1. Relationships among system-call, meta-filesystem and filesystem operations

3.2. How RFS Handles These Issues in System V, Release 3

In System V, Release 3, RFS is implemented using the system call abstraction. There is one RFS operation for each standard UNIX system call involving file operations and these operations take the same arguments as do the corresponding system calls. Thus, just as is true for the system call interface, some RFS operations take pathnames and others take file handles that represent resolved references to remote files. Some RFS operations, (e.g., RFS_OPEN) take pathnames as arguments and return file handles as results. The RFS operations are listed in Table 1.

Operation	Action
RFS_ACCESS	Check access permissions
RFS_SYSACCT	Do system accounting
RFS_CHDIR	Change directory
RFS_CHMOD	Change file mode
RFS_CHOWN	Change file owner
RFS_CHROOT	Change root directory
RFS_CLOSE	Close a file
RFS_CREAT	Create a file
RFS_EXEC	Exec a file
RFS_EXECE	Exec a file with an environment
RFS_FCNTL	File control
RFS_FSTAT	Stat a file (uses file descriptor)
RFS_FSTATFS	Stat a filesystem (uses file descriptor)
RFS_IOCTL	Ioctl
RFS_LINK	First half of link() operation
RFS_LINK1	Second half of link() operation
RFS_MKNOD	Make a device file
RFS_OPEN	Open a file
RFS_READ	Read from a file
RFS_SEEK	Seek on a file
RFS_STAT	Stat a file (uses pathname)
RFS_STATFS	Stat a filesystem (uses pathname)
RFS_UNLINK	Unlink a file
RFS_UTIME	Change times on file
RFS_UTSSYS	ustat()
RFS_WRITE	Write a file
RFS_GETDENTS	Read directory entries in a filesystem-independent format
RFS_MKDIR	Make a directory
RFS_RMDIR	Remove a directory
RFS_SRMOUNT	Server side of remote mount
RFS_SRUNMOUNT	Server side of remote unmount
RFS_COREDUMP	Dump core
RFS_WRITEI	Internal form of write system call
RFS_READI	Internal form of read system call
RFS_RSIGNAL	Send remote signal
RFS_SYNCTIME	Synchronize time between machines
RFS_IPUT	Free a remote inode
RFS_IUPDATE	Update a remote inode
RFS_UPDATE	sync() a remote filesystem

Table 1. RFS operations.

To implement the RFS server in terms of this model, the UNIX system call interface already provided within the kernel was employed. Thus an RFS client request is carried out on the server by a kernel-only server process which directly calls the system call entry-point routine corresponding to the request. However, there are some problems in adapting this in-kernel system call entry-point interface for uses other than its original purpose. One difficulty concerns the arguments provided for the interface operations. These arguments not always adequate for the purposes of RFS; thus, a number of global parameters (e.g., those found in the user structure) were added to remedy this deficiency. Another problem arises with operations involving data movement across the interface. Such operations do not accept or return data in a manner sufficiently flexible for RFS. Data is copied to or from the user address space of the process doing the system call operation. Because the designers of RFS wanted the server to reside entirely within the kernel, they were forced to modify the data movement routines to recognize if the system call was being made on behalf of RFS and take appropriate action.

Other significant complications arose in the process of pathname resolution. As discussed above, RFS subscribes to the mount abstraction in order to preserve UNIX file naming semantics. This means that the client first detects the remoteness of a pathname when it crosses a remote mount point in the midst of parsing the name. Unfortunately, the client has already descended below the system call interface layer at this point and is performing meta-filesystem operations. To solve this problem, the generic pathname parsing routine in System V, Release 3 was modified to recognize remoteness, switch out to execute the RFS remote call corresponding to the local system call already in progress, and then *longjmp()* out of the local system call code on return.

A further complication with pathname resolution can occur on the server. Just as the client must deal with the fact that a system call may go remote in the middle of parsing, the RFS server must handle the case where a request received from a client goes *local* in the middle of parsing, because of a "." in the pathname. In this case the RFS server must abort the parse and return the remainder of the pathname to the client to continue evaluation.

It is because RFS is a system-call based protocol and yet at the same time behaves as a filesystem under the mount model, that it sometimes finds itself halfway between the two abstractions, patching up meta-filesystem operations.

3.3. How RFS Handles These Issues in SunOS

To resolve these problems in SunOS, RFS is implemented using the vnode/VFS architecture. The RFS client-side is implemented as a vnode/VFS filesystem type. This eliminates any requirements for modifications to the generic pathname resolution code or for the use of a *longjmp()*, as RFS operations are then performed only in the process of carrying out filesystem-specific operations at the appropriate place and time in generic system call code. On the server side, RFS is implemented using vnode/VFS operations, rather than system-call entry points. Because vnode/VFS operations were designed from the outset for generic use, they proved quite sufficient to meet the needs of the RFS server implementation. Thus no modifications were required to generic kernel code or data structures solely to support RFS.

4. Client-Side Implementation

Because of the mismatch between the filesystem abstraction provided by the vnode/VFS layer and the system call abstraction on which RFS is based, implementing the RFS client as a vnode/VFS filesystem type proved to be a substantial challenge. The operations of interest provided by the vnode/VFS interface are listed in Table 2.

A key requirement of this interface is for a *vn_lookup()* operation that takes a component of a pathname and a file handle to the directory in which that component resides, and returns a file handle to the file which the component represents. In order to be congruent with RFS semantics, this operation should also cause the server on which the file resides to hold the file's inode on behalf of the client. This poses a difficult problem, because there is no one operation in the system call interface that is intended to accomplish this. Potential candidates include operations such as RFS_OPEN and RFS_CHDIR, however, these operations will fail in some cases where a lookup op should succeed and they may have other undesired side effects on the server.

Operation	Action
<code>vfs_mount()</code>	Mount a filesystem
<code>vfs_unmount()</code>	Unmount a filesystem
<code>vfs_statfs()</code>	Return filesystem statistics
<code>vfs_sync()</code>	<code>sync()</code> a filesystem
<code>vn_open()</code>	Open a file
<code>vn_close()</code>	Close a file
<code>vn_rdwr()</code>	Read/write a file
<code>vn_ioctl()</code>	Do <code>ioctl</code> on a file
<code>vn_getattr()</code>	Get file attributes
<code>vn_setattr()</code>	Set file attributes
<code>vn_access()</code>	Check access permission
<code>vn_lookup()</code>	Lookup a file, return vnode
<code>vn_create()</code>	Create a file
<code>vn_remove()</code>	Remove a file
<code>vn_link()</code>	Link a file to a new name
<code>vn_rename()</code>	Rename a file
<code>vn_mkdir()</code>	Make a directory
<code>vn_rmdir()</code>	Remove a directory
<code>vn_readdir()</code>	Read directory entries
<code>vn_lockctl()</code>	Lock a file
<code>vn_inactive()</code>	Release a vnode

Table 2. Vnode/VFS operations.

Fortuitously, RFS does contain an operation which has the desired characteristics. The `link()` system call is accomplished in RFS by breaking the call into two parts, `RFS_LINK` and `RFS_LINK1`. This is required because `link()` takes two pathnames, and after the first name is evaluated on the server, one must return to the client to initiate the evaluation of the second pathname. Accordingly, the `RFS_LINK` operation does nothing except evaluate a pathname on the server, hold the inode corresponding to the pathname, and return a file handle (which is actually a cookie for the remote inode) for the file. Thus, this operation has exactly the characteristics desired of a lookup operation.

A second major problem associated with the abstraction mismatch between filesystem operations and system call operations involves arguments to the operations. Most vnode/VFS operations take one or more vnodes as arguments. These vnodes need to contain in their private data portion an RFS file handle to the file of interest; this file handle can then be supplied as a parameter to execute the RFS calls necessary to accomplish the desired vnode/VFS operation. The question is, then, what is an appropriate file handle to be provided to RFS? As has been discussed, some RFS operations are name-based whereas others are based on file descriptors. And, in general, when a vnode is created for an RFS remote file, there is no known way of knowing whether it will be used for a name-based operation, a file-descriptor-based operation, or both.

The solution to this problem is to store *two* file handles to the file in the private part of each RFS vnode; then, either one can be used as needed. One is the cookie for the remote inode obtained from doing a remote lookup on the name. The second consists of a remote inode cookie for the parent directory in conjunction with the component name of the file in that directory. Happily, these parameters are all available when the vnode is created in the `vn_lookup()` routine.

Storing a file name in a vnode, however, creates some new complications. The relation between file names and files is many-1, whereas between vnodes and files it is 1-1. Storing file names in vnodes thus implies the existence of multiple vnodes for the same file, each with its own attached name. This raises the question of whether it is reasonable to break the 1-1 relationship between vnodes and files.

One place where this becomes a problem is as follows. There are several places in the kernel where the identity of vnodes is tested with an equality test, i.e.,

```
if (vnodeptr1 == vnodeptr2)
```

Although the explicit semantic of this test is, do these pointers have the same value, its implicit semantic is, do these vnodes refer to the same file? Clearly, with multiple vnodes for the same file such a test might fail where it should succeed. To solve this problem, a new vnode operation, *vn_cmp()*, was added to the vnode/VFS interface, that compares two vnodes to see if they refer to the same file; kernel tests for vnode equality were then modified to use the *vn_cmp()* operation. RFS implements this operation by comparing the remote inode cookies in the two vnodes, to see if they refer to the same remote file. For those filesystems that do not make use of multiple vnodes, this test simply defaults to the one shown above.

The other problem that occurs when breaking the 1-1 relationship between vnodes and files, has to do with state in the vnode. If different vnodes refer to the same file, then the state in those vnodes should always be consistent with one another. A complete solution to this problem would require something like a common state structure shared among all vnodes referring to the same file. However, this seemed too drastic an alteration to the vnode/VFS layer just for the purposes of accommodating RFS. Instead a partial solution was adopted. When a new RFS vnode is created, the cache of existing RFS vnodes is checked. If there is a node in the cache that refers to the same file as the new one and has the same name, then this node is returned. If a node in the cache refers to the same file but has a different name, then a new node is created with the new name, and the state information is copied from the cached node to the new one. While this does not represent a complete solution, it is sufficient, in conjunction with the *vn_cmp()* test, to prevent anomalous behavior in RFS.

Two other points that had effects on the vnode/VFS interface are worthy of mention. The *vn_close()* operation which was formerly called only on last close of a file descriptor is now called every time with a reference count. This conforms to the semantic of the RFS_CLOSE call. RFS requires this semantic so that remote locks held on behalf of a client may be released properly. Other filesystem types that only wish to do real work on last close can simply check the reference count and return.

A second point concerns the *vn_getattr()* operation, which returns the attributes of a file. One such attribute is a filesystem id that is used to uniquely identify the filesystem in which the file resides. This value is returned by the *stat()* system call and used by utilities such as *pwd* to uniquely identify files. Traditionally in the UNIX operating system, this id contains the major and minor device number of the physical device on which the filesystem resides. A number of utilities make use of this fact to find the device corresponding to the filesystem in which a given file resides. It is necessary to insure that different filesystem types do not conflict in their use of the name space of the filesystem id. To accomplish this, a policy of name space usage was implemented that must be followed by all filesystem types. Basically this policy allows filesystem types that reside on local devices to use the device number as a filesystem id. Other filesystem types must use a macro to construct the id, and this macro prevents name space conflicts.

Once the major problems discussed above had been solved, the remainder of the client-side implementation consisted simply of writing the vnode/VFS operations for the RFS filesystem type in terms of RFS operations. To facilitate this, a layer of RFS operation routines was written, each of which accepts arguments for one RFS remote call, performs the call and returns the results. Using these routines, the vnode/VFS operations were implemented in a straightforward manner. The correspondence between vnode/VFS operations and the RFS operations used to implement them is shown in Table 3.

It is apparent from Tables 1 & 3 that many vnode/VFS operations required more than one RFS call to implement, and that many RFS calls were not needed to construct the functionality required by the vnode/VFS layer. This suggests that the set of operations provided by RFS is redundant in terms of the basic building blocks required to implement a filesystem interface.

5. Server-side Implementation

As discussed above, the use of the in-kernel system call interface by the RFS server necessitated many changes to generic kernel system call code and data structures. To eliminate the need for these modifications, the SunOS RFS server was implemented using the operations provided by the vnode/VFS

Vnode/VFS Operation	RFS Operations Used
vfs_mount()	RFS_SRMount, RFS_STATFS
vfs_unmount()	RFS_SRUMOUNT
vfs_statfs()	RFS_STATFS
vfs_sync()	RFS_UPDATE
vn_open()	RFS_OPEN
vn_close()	RFS_CLOSE
vn_rdwr()	RFS_READ, RFS_WRITE
vn_ioctl()	RFS_IOCTL
vn_getattr()	RFS_FSTAT
vn_setattr()	RFS_OPEN, RFS_CHMOD, RFS_CHOWN, RFS_UTIME
vn_access()	RFS_ACCESS
vn_lookup()	RFS_LINK
vn_create()	RFS_OPEN, RFS_MKNOD, RFS_LINK
vn_remove()	RFS_UNLINK
vn_link()	RFS_LINK, RFS_LINK1
vn_rename()	RFS_LINK, RFS_LINK1, RFS_UNLINK
vn_mkdir()	RFS_MKDIR, RFS_LINK
vn_rmdir()	RFS_RMDIR
vn_readdir()	RFS_GETDENTS
vn_lockctl()	RFS_FCNTL
vn_inactive()	RFS_IPUT

Table 3. Vnode/VFS operations and RFS operations used to implement them.

layer. This interface proved sufficiently flexible and general to accommodate the RFS server's needs without modification. Some details of the implementation are described below.

Because of the remote system call model, the RFS server code must be prepared to parse a complete UNIX pathname, rather than only evaluating one component as is the case for NFS. Furthermore, as previously described, the pathname evaluation code must be prepared to abort the parse and return the remainder of the pathname to the client if a ".." causes the evaluation to cross back over the server mount point. To accomplish this, a *lookupname()* routine quite similar to that used by the local system call code, was written for the RFS server. This routine parses pathnames according to the standard UNIX semantics, but deals appropriately with the crossing of RFS server mount points. Any client request using a pathname is serviced by first calling this routine to evaluate the name.

The server was cast in a more object-oriented style than in the original implementation, by having a vector of function calls, consisting of one procedure for each RFS system call. The main server routine simply unpacks client requests and switches out to the appropriate procedure based on the client request opcode.

Much server-process administrative information, formerly kept in the user structure and the process table, was removed and placed in a parallel server-process structure that is allocated whenever a new kernel server is forked and de-allocated when it exits. Removing this information from generic kernel data structures means that RFS can be configured out of the SunOS kernel (in the same manner as NFS and the 4.2 BSD filesystem) without leaving its vestiges around. The server-process structure contains information such as the process and machine id of the client being served, the server inode currently being manipulated on the client's behalf, and a pointer to the virtual circuit over which communication with the client is taking place. Server-process scheduling is administered via linked lists of server-process structures.

6. Recovery

While many changes were made to RFS code to avoid impacting SunOS, one important goal of the implementation was that the SunOS version of RFS would provide the same functionality and semantics as in System V, Release 3. Accordingly, the SunOS RFS server holds open vnodes and maintains other state on behalf of clients. Because of this, recovery code is needed to return to a reasonable state when a connection goes down. The RFS recovery code was ported over from System V, Release 3, essentially intact.

7. Administrative Support

RFS administrative support, including authentication, name service, connection service, and miscellaneous utilities, was ported to SunOS with few changes. Exceptions to this were the *mount* command and system call which were integrated with existing SunOS code. Future directions might include integration of various RFS administrative facilities with corresponding existing Sun facilities (e.g., integrating the RFS name advertise service with Sun's Yellow Pages¹⁰).

8. Heterogeneity Issues

To support communication between heterogeneous machine and operating system types, RFS makes use of Sun's eXternal Data Representation format (XDR). When an RFS circuit is setup between two machines they first exchange information to determine, among other things, if they have the same machine type. If they do not, then all further communication takes place using XDR. One of the original intentions of this project was to demonstrate that RFS could function between heterogeneous machines running different operating systems, since this had not been previously shown.

During the port, some problems involving RFS's use of XDR and the passing of structured data were encountered that required modifications to System V, Release 3 code. Some of these occurred in the XDR implementation and involved, for example, not allocating enough buffer space for an XDR conversion. Others related to the passage of information over the wire about the size or address of data on one machine which were then interpreted and used on another machine.

One example of this occurs with a field passed in RFS remote calls involving data movement. This field contains a pointer into the client process's memory address space, and in the course of the data movement protocol this address is passed to the server, incremented by the server and then passed back and used by the client. Fortunately, it turns out that there is no need for the client to make use of the value returned by the server; the client can simply keep track of the address pointer itself.

Another example involves the passage of directory entry data over the wire. Because this data is structured, it is not possible to send size values over the wire and have them be meaningfully interpreted at the other end. However, the System V, Release 3 implementation passes and uses fields containing information about the sizes of individual directory entries, and the total size of a chunk of directory entries. Once again it is possible to circumvent these problems by ignoring the received values and re-computing them manually from the data received.

9. Conclusions

In addition to providing RFS functionality in SunOS, this prototype has demonstrated that an RFS can be constructed that functions between heterogeneous machine types and that it need not require extensive modifications to generic kernel code to support its implementation. In addition, this prototype operates between UNIX System V, Release 3 and the 4.2BSD-derived SunOS. The issues that arose during the course of this implementation shed some interesting light on system call and filesystem abstractions and the limits to which these abstractions can be pushed in the pursuit of design goals.

10. Acknowledgements

I would like to thank the members of the Systems Software Group at Sun, in general, for their input and advice during this project. Solutions to most of the hard issues evolved out of fruitful discussions with many people including Steve Kleiman, Jon Livesey, Don Cragun, Bob Lyon, Guy Harris, Bill Shannon, Glenn Skinner, Daniel Steinberg and Joseph Moran.

11. References

- 1 A.P. Rifkin *et al*, "RFS Architectural Overview", *USENIX Conference Proceedings*, Atlanta, Summer, 1986.
- 2 R. Sandberg *et al*, "Design and Implementation of the Sun Network Filesystem", *USENIX Conference Proceedings*, Portland, Summer, 1985.
- 3 S. Kleiman, "Vnodes: An Architecture for Multiple Files System Types in Sun UNIX", *USENIX Conference Proceedings*, Atlanta, Summer, 1986.
- 4 M. Rosen, M.J. Wilde, and B. Fraser-Campbell, "NFS Portability", *USENIX Conference Proceedings*, Atlanta, Summer, 1986.
- 5 B. Lyon, "Sun External Data Representation Specification", Sun Microsystems, Inc. Technical Report, 1984.
- 6 *AT&T UNIX System V STREAMS Programmer's Guide*, AT&T, 1986.
- 7 *AT&T UNIX System V Network Programmer's Guide*, AT&T, 1986.
- 8 R. Rodriguez, M. Koehler, and R. Hyde, "The Generic File System", *USENIX Conference Proceedings*, Atlanta, Summer, 1986.
- 9 M.J. Karels and M.K. McKusick, "Towards a Compatible Filesystem Interface", U.C., Berkeley Technical Report, 1986.
- 10 P. Weiss, "Yellow Pages Protocol Specification", Sun Microsystems, Inc. Technical Report, 1985.

GFS Revisited

or

How I Lived with Four Different Local File Systems

Matt Koehler
mjk@DECVAX.DEC.COM

ULTRIX¹ Engineering Group
Digital Equipment Corporation
Merrimack, New Hampshire 03054

1. INTRODUCTION

The Generic File System (GFS) interface was designed and implemented in the spring of 1986. A more complete reference of GFS can be found in (Rodri1986a). Once the interface was implemented, we were interested in determining its completeness. To achieve this, several widely used file systems were prototyped using the interface. These file systems were: UFS (4.3BSD's Fast File System), System V file system, the MS-DOS² file system, and ODS-II (native VAX/VMS file system).

The on-disk structures of each of the file systems were studied so that the maximum number of UNIX³ (read 4.3BSD) semantics could be implemented.

Each architecture was further examined to understand performance implications of disk and file system layout and how that layout affects the file system implementation. Finally, the needs of NFS⁴ were contrasted with the services provided by each file system architecture.

For the remainder of this paper, UNIX and UNIX file system semantics will refer to 4.3BSD and its file system functionality.

2. SYSTEM ENVIRONMENT

The system used for development was a MicroVax-II with seven megabytes of physical memory. The disk subsystem consisted of an RQDX3 controller and an RD54 (160 megabyte) disk drive.

The operating system was ULTRIX Version 2.0 configured with 10% (700K) of memory devoted to the buffer cache. The base operating system differs from V2.0 in that four prototype file systems were added. These additions required only the changing of a data and configuration file. Each of these prototypes took advantage of 4.3BSD's namei caching, LRU gnode caching, and buffer caching. The prototypes implemented the full set of 4.3BSD file system related system calls when ever possible. For example, the System V prototype included code to perform a mkdir system call.

3. THE ODS-II PROTOTYPE

ODS-II (also known as Files-11) is the file system that the RSX and VAX/VMS operating systems use. It is a prevalent file system in the mini-computer marketplace. The ODS-II prototype does not include code for writing or creating files, or modifying directories.

The ODS-II file system is constructed from volumes. Each volume may be an entire disk, many loosely coupled disks, or many tightly coupled disks. When the volume is constructed of loosely coupled disks, each disk may become a volume on its own. Therefore, not all disks in a loosely

¹ ULTRIX, MicroVAX, VAX, DEC, and VAX/VMS are trademarks of DIGITAL.

² MS-DOS is a trademark of the Microsoft Corporation

³ UNIX is a registered trademark of A. T. & T.

⁴ NFS is a registered trademark of Sun Microsystems Inc.

coupled volume need to be present for the volume to be mounted and used. The tightly coupled disk appears as a concatenation of all the disk media into a single disk address space.

The architecture supports file system block sizes ranging from 512 bytes to 64 kilobytes in 512 byte increments. The format also supports file names of up to 86 characters in length (not including revision numbers). The name space includes all alpha-numeric characters, '_', and '\$'. While the prototype includes most of the ASCII character set, creating files with some characters may create difficulty when transporting the ODS-II file system to other operating systems.

3.1. ODS-II Architecture

All disk blocks within an ODS volume appear within the file system. The boot block and file system control files appear in files with well known identifiers and locations. There are sixteen reserved file identifiers for use by the operating system. The ODS-II specification defines nine of the identifiers.

The first reserved file is **INDEXF.SYS**. The index file resides in the front of the disk and contains a system boot block, a home block, file headers (which are described later), and a file header free map. The home block contains information about the location of other reserved files, the device type, the file system block size, and volume identification and protection information.

The second reserved file, **BITMAP.SYS**, is the storage bitmap file. This file contains a map of all blocks on the system. Each bit within the map indicates whether the corresponding file system block is free. The third file, **BADBLOCK.SYS**, holds all known bad blocks on the device. The fourth file, **000000.DIR**, is the master file or root directory. The fifth file, **CORIMG.SYS**, is the kernel crash image (for example, vmcore.0).

The sixth file is **VOLSET.SYS**. This file defines the relationship between disks in a tightly coupled multi-disk set. The seventh file is **CONTIN.SYS**. This file describes loosely coupled multidisk volumes.

The eighth file is **BACKUP.SYS**. It is used to log and control backups. The ninth file is **BADLOG.SYS**. Disk blocks that the operating system suspects are becoming bad are listed (but not held) here.

3.1.1. ODS-II Directory Description

ODS-II directories are contiguous files constructed of variable length directory entries. A directory entry contains the size of the entry, the maximum number of versions allowed for this name, a flag for describing the directory entry, the file name and its length, and a mapping of version numbers to file identifiers. It should be noted that version numbers need not be continuous. Most implementations of ODS-II allow individual versions of a file to be removed.

The structures held within the directory file contain only the children of the directory. Neither a self-referential file ('.') nor a file referencing the parent ('..') exists.

3.1.2. ODS-II File Structure Description

The attributes of a file are described by the **FILE HEADER**. This header is somewhat equivalent to UFS's on-disk inode and is a common point for providing information about the file. There are four structures to a file header. They are:

3.1.2.1. ODS-II File Header

The first structure is a header containing information to check the validity and accessibility of the file. The major components of this structure are:

- Indices to the remaining three substructures
- File record attributes
- Protection attributes
- File characteristics
- Ownership
- Backlinks
- Length

The file record attributes field contains information which describes the structuring of records within the file. There are several defined record formats. Records may be of fixed or variable length; they may be stream oriented with record terminators being carriage returns, line feeds, form feeds or combinations thereof. There is also the ability to have no record information at all. This is analogous to stream oriented files under the UNIX file system.

Protection modes are available on four operations: read, write, execute, and delete. In turn, this protection scheme applies for a user, a group, the world, and the system (root). The protection may also include restrictions to a particular VAX processor mode (user, supervisor, exec or kernel).

There are a myriad of file characteristics. Files may disable backups of themselves. They may change the write strategy to write back or write through. Files may request that reads are followed by a read/compare. They may likewise request that write operations are followed by a read/compare. Files may ask that through best effort, blocks are allocated contiguously. Also files may request that when a block is deleted, the deleted disk block will be overwritten.

The ownership of a file is described by a user identification code (UIC). The UIC is similar to uid and gid used by UNIX systems. The length field is the size of the file in bytes. Finally, the backlink is a pointer to the parent directory.

3.1.2.2. ODS-II File Identification Header

The file identification header contains identity and accounting information about the file. The major components of the file identification header are:

- File name
- Revision number
- Create, revision, expiration, and backup dates

3.1.2.3. ODS-II Map Header

The map header contains map structures allowing file blocks (VBNs) to be transformed into disk blocks (LBNs). The map structures describe the number of contiguously allocated blocks from a starting disk block. Three map types exist allowing up to 2^{30} contiguous blocks to be mapped from a starting block represented in up to 2^{32} bits. Different map types may be intermixed within the same map header. The mapping strategy allows files to have holes or unallocated spaces.

3.1.2.4. ODS-II Access Control Lists

The access control lists (ACLs) are an optional part of the file header. The ACLs hold data permitting exceptions to the permissions described in the ODS-II file header. It is interesting to note the ACLs may be used to permit or deny access to a file for a group or individual.

3.2. ODS-II Prototype Design and Implementation

Since the ODS-II file header is a crucial element for describing a file, it is necessary to associate the header with a gnode. There is space reserved in a gnode for file system implementations to store file information. The space is 88 bytes long; file headers contain 512 or more bytes. Therefore, each *ods_gget* must allocate additional space for the header and attach that space to the gnode. To allow LRU gnode caching, the gnode must preserve the reference to the file header. It becomes necessary to recover the header space when the gnode is reused for another file. Adding a mechanism for header space recovery to the GFS interface allows the LRU gnode cache to work.

3.2.1. ODS-II VBN to LBN Mapping

Another functionality requiring thought is *ods_bmap*, or the mapping of virtual block numbers (VBNs) to logical block numbers (LBNs). With the exception of ODS-II, all file system prototypes have an array of file block indices permitting one to one VBN to LBN transformations.⁵ A map structure in ODS-II maps a range of VBNs. This causes *ods_bmap* to iterate through each map structure until the VBN is found.

3.2.2. ODS-II Pathname Translation

The lack of '.' and '..' causes *ods_namei* to special case these names and return valid information. Likewise, *ods_getdirent* needs to return '.' and '..' when returning directory information. Further, the file **000000.DIR** is the root directory for the file system. Since **000000.DIR** and '.' refer to the same directory, **000000.DIR** cannot appear in the name space without causing looping in file system traversals.

Finally, providing code for VBN allocation (that is, extending the file) is complicated by strategies for contiguous files and by multi-disk volumes.

3.3. ODS-II Performance

Since the ODS-II prototype does not support write, it was necessary to use an existing file to measure read performance. **INDEXF.SYS** is the largest file on the disk containing just more than 2 megabytes. This file was read repeatedly asking for a varying number of bytes. The following table compares ODS-II performance to reading blocks from the character device.

Read Performance (in kB/sec)							
	512b	1K	4K	8K	16K	32K	64K
ODS-II	15	31	173	204	280	350	388
Device	25	42	132	184	257	320	357

To measure ODS-II performance with read requests greater than 8K, it was necessary to rebuild the kernel to change the maximum buffer size to 64K. It was also necessary to change *ods_rwgp* to override the clustering factor.

ODS-II attains better file system throughput by increasing the file system block size. Binaries and other large files may be well hosted on an ODS-II volume.

3.4. ODS-II Supporting NFS

There are no architectural limitations restricting the use of ODS-II as a file system served by NFS. After fixing a few implementation problems, NFS worked perfectly.

⁵ Calling an MS-DOS FAT chain an array is stretching the analogy. see section 4.1.2 for a detailed explanation.

3.5. ODS-II Strengths

The ODS-II architecture is rich with functionality.

There are several file attributes that are not present within any of the other file system architectures. Also, the ability of file systems to span disks is attractive. Finally, the file system performance is good. Allowing 64K blocks reduces the number of transactions handled by a disk controller.

3.6. ODS-II Limitations

The inclusion of delete and an additional grouping by system causes ODS-II file permissions to be not completely representable using UNIX file system semantics.

Only two types of files (regular files and directories) are defined in the ODS-II architecture. Block and character special files, soft links, and named FIFOs cannot be created or accessed. Since each file header has a pointer back to its parent directory, hard links cannot be created.

The lack of '.' and '..' in directories causes problems. While the functionality associated with these files can be emulated, their names are not reserved in the ODS-II file name space.

Finally, ODS-II allows for file system block sizes to be very large. Since there is no concept of fragmentation, a one byte file on a 64K block file system consumes 64K.

4. THE MS-DOS PROTOTYPE

The popularity of the IBM⁶ personal computer causes the MS-DOS file system to be one of the most prevalent file system architectures in existence.

The MS-DOS architecture supports file system block sizes of 512 bytes, 1K, or 2K. The 2K file system block size has been added for support of 20M hard drives. File names can have a length of 11 characters which is broken into an 8 character base and a 3 character extension. The base and extension characters are separated by a '.' (which is not stored in the file name). Even though only the names '.', '..', and names beginning with '\0345' or containing '\0' are reserved, most implementations only allow upper case letters, numbers, and a few symbols in the name space.

4.1. MS-DOS Architecture

An MS-DOS file system contains four components: the boot record, the FAT, or file allocation table, the root directory, and file system data blocks. The FAT is usually replicated once for reliability.

4.1.1. MS-DOS Boot Record

The beginning of the boot block contains an index to the boot record. The boot record holds data concerning the format and size of the disk. Included in this record is the size of a data cluster (the file system block size), the number of root directory entries, and the number of sides and heads on the media. An operating system ID, the size and number of copies of the FAT, and the number of system reserved sectors are also included in the boot record.

4.1.2. MS-DOS File Allocation Table

The FAT contains the mappings of VBNs to LBNs. There is a FAT entry for each data block on the file system. FAT entries appear in two formats: 12 bit (1.5 bytes) and 16 bit. The 12 bit format is the most common. 16 bit FATs were introduced with support for 20M hard disks.

Allocated FAT entries are members of a data block chain. Since each entry identifies a data block, determining the value held in the current entry yields the next entry in the FAT chain (and hence next data block). See section 4.1.4 for further information.

⁶ IBM is a registered trademark of International Business Machines

FAT entries zero and one are reserved. Entry zero contains a media identifier. Entry one is unused. All other entries contain either an index to the next entry in the FAT chain, a flag indicating the end of the chain (entries with the high 9 or 13 bits set), a bad block flag (0xFF7 for 12 bits, 0xFFF7 for 16 bits), or unallocated block flag (0).

4.1.3. MS-DOS Directory Description

MS-DOS directories are files constructed from not necessarily contiguous blocks containing fixed length directory entries. Each directory entry is 32 bytes long and is defined as follows:

```
struct msdos_dir {
    u_char    md_name[8];
    char      md_ext[3];
    u_char    md_attr;
    char      md_fill[10];
    u_char    md_tod[4];
    u_char    md_cluster[2];
    u_char    md_size[4];
};
```

Each of these fields will be described in section 4.1.4.

As expected, there is one directory entry for every file in the directory. With the exception of the root directory, the first two entries in each directory are for files '.' and '..'. If the first character of the file name is '\0345', then the slot is unused. If the first character is '\0', then the end of the directory has been reached.

The root directory is handled differently from its sub-directories. First, the files '.' and '..' do not exist. This is reflected by the fact that the root directory resides in a reserved area on disk. Further, the root directory is a static set of contiguous blocks causing there to be a fixed number of directory slots.

The directory entry contains data concerning file attributes. Since directories (other than the root of the file system) have two entries (one naming the directory, the other '.'), there are two sets of attributes for each directory. As will be discussed in the implementation section, this causes problems for updating directory attributes.

4.1.4. MS-DOS File Description

The attributes of a file are described in the file's directory structure. Since only 8 bits are used for protection, security is minimal. The protection allowed is:

- Read only
- Hidden — analogous to file names beginning with '.' in UNIX
- System — a remnant of CP/M
- Volume label — refers to the name of the volume not to a file
- Subdirectory
- Archive — indicates if the file has been backed up but not modified

The time stored in the directory entry is the creation or last modification time. It is a binary coded 32 bit value. Its structure is as follows:

```
struct msdos_tod {
    unsigned    sec        : 5;
    unsigned    min        : 6;
    unsigned    hour       : 5;
    unsigned    day        : 5;
    unsigned    month      : 4;
    unsigned    year       : 7;
};
```

Since only 32 discrete values can be stored in the second field, the granularity of file creation time is in units of 2 seconds. The year field holds the number of years since 1980.

The starting cluster field in the directory holds the head of the FAT chain. If the cluster was 2 and we use the FAT chain shown in figure 1, the data blocks 2, 3, and 5 belong to the file.

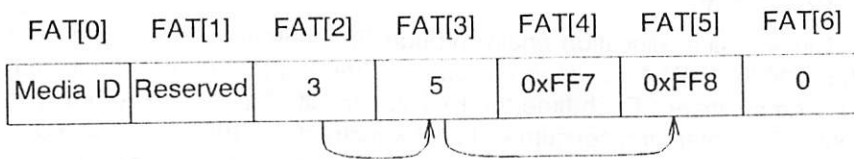


Figure 1

The file size is a 32 bit quantity. This may not reflect the true size of the file. First, MS-DOS directories do not store their size. Second, in an MS-DOS environment, the character 'Z' marks the end of text file. Since these text files contain an end of file marker, zero-length files do not exist.

4.2. MS-DOS Prototype Design and Implementation

While the structures associated with the MS-DOS file system are simple to understand, their layout causes problems. Since the initial target architecture for the MS-DOS file system was an Intel⁷ 8088 microprocessor, there was no concern for quantities crossing four byte boundaries. For example, the create time in the directory structure is a 32 bit quantity and its address is 22 bytes into the directory. Likewise, the starting cluster and size cross long word boundaries. This caused every directory encode and decode routine to access all fields as characters and reconstruct them (fortunately, everything is stored in VAX order!).

Because the directory entry is only 32 bytes long, each directory is held within the file system specific part of a file's gnode. This provides a big performance gain when the file can be found in the gnode cache.

4.2.1. MS-DOS File Identification

Another piece of the file system architecture affecting the prototype design was the lack of a file identifier number. Because many UNIX tools require a somewhat unique "on-disk inode number", an algorithm was needed to create file IDs. Since an LRU gnode cache was being used, file IDs must be unique. Also these file IDs must be consistent for every instance of a file. Remember that a directory has two names and attribute structures.

The following algorithm was created for identifying files. If the file in question is not a directory, then the file ID is the starting cluster of the file's parent directory shifted left 10 bits or'd with the

⁷ Intel is a registered trademark of Intel Corporation

directory slot number. For example, if the starting cluster for the parent directory is 2 and the file was found in slot 33, the file ID would be 2081 ($2 \ll 10 + 33$).

This scheme works until it is necessary to update directory attributes. The file ID for a directory is simply its starting cluster shifted left 10 bits. To allow *msdos_gupdat* to update both incarnations of a directory, the old file ID (that is, the ID found by or'ing the directory slot number with the shifted block number of the parent) is stored in a file system specific area in the gnode. This modification of the algorithm allows the pathname *X* and *X/.* to produce the same file ID. The root directory causes still one more special case. Since '.' and '..' are not present in the root directory, the root file ID is 2. Note that the root file ID is unique because MS-DOS reserves the first disk block for the boot record,

4.2.2. MS-DOS Pathname Translation

Because the root directory contains no self referential files, it is necessary to special case *msdos_namei* to trap the names '.' and '..'. Also, when a directory is found, *msdos_namei* must store the old file ID and obtain a new set of attributes. This creates some ugliness in the *msdos_namei* code.

4.2.3. MS-DOS VBN to LBN Mapping

A file's blocks are mapped in a space allocation chain through FAT entries. To find the LBN associated with VBN 2, the LBN for VBN 1 needs to be found. Therefore, determining a logical block from a virtual block becomes linear. Each time the FAT is consulted, a 12 bit FAT entry is converted to a 32 bit quantity. This mapping consumes 12 VAX instructions (not including register setups). Likewise, mapping a 32 bit quantity to a 12 bit FAT entry uses 19 VAX instructions. Decoding a FAT chain is more expensive than translating an MS-DOS file name.

There are several methods for reducing the time needed to decode a FAT chain. Since decoding VBN 2 requires knowing the LBN associated with VBN 1, storing information about the last decoded VBN/LBN pair can reduce the FAT search time. This presupposes that files are read sequentially. When a file is being truncated, blocks are removed from the end of the file. Therefore, storing the last VBN/LBN pair can return to linear search time.

The entire FAT chain can be decoded when the directory entry is retrieved from disk. Since the directory entry is cached while inactive, decoding the FAT chain at name resolution may be an acceptable solution.

At the time this paper was written, the MS-DOS prototype stored the most recently used VBN/LBN pairs. Because only read and write performance was measured, this algorithm was sufficient for measuring expected file system performance.

4.2.4. MS-DOS Block Allocation

I was unable to locate a description of how MS-DOS systems allocate blocks to files. Examining files on an MS-DOS system indicated that there is no allocation policy. Blocks within a file were scattered throughout the device. Since many MS-DOS systems depend heavily on diskette drives, this scattering can cause dismal file system performance.

4.3. MS-DOS Performance

To assess read and write performance, a 4 megabyte file was created on the MS-DOS file system. This file was read and written using different block sizes. Measured performance follows:

I/O Performance (in kB/sec)			
	512b	1K	8K
Write	8.3	7.4	7.5
Read	26	26	26

Even though the *read* and *write* system calls could request I/O in units larger than the file system block size, *msdos_rwgp* breaks the I/O request down into file system block size pieces. Therefore, while the number of system calls was reduced by a factor of 16, performance is not expected to improve greatly since the kernel still posts the same number requests to the disk.

Write performance was poor because *msdos_bmap* must decode a 12 bit FAT entry when locating a free block in the FAT and when traversing the file's FAT chain. Better management of file block lists and free block lists should increase performance dramatically.

4.4. MS-DOS Support for NFS

The lack of an adequate file identifier caused the MS-DOS file system to be initially unusable with NFS. After solving the file ID problem, NFS coexists with MS-DOS.

A significant loss in performance was measured over NFS. Small block sizes may cause more packets to be transmitted between NFS server and client.

MS-DOS files cannot have holes. Currently, it is possible to create a UNIX file, seek well past the end of the file and write a block. This causes the file to have unallocated VBNs. Since MS-DOS file blocks are constructed from a FAT chain, no holes can exist within a file. Finally, NFS uses a file generation number to indicate whether a file has changed identity (by being removed). MS-DOS provides no method of storing this generation number.

4.5. MS-DOS Strengths

Since MS-DOS systems are widely used, the MS-DOS file system can be a good file transport between machines. Allowing such file systems to be read and written under UNIX systems permits MS-DOS machines to have indirect access to many of UNIX's facilities.

The MS-DOS file system also consumes little disk space for file system overhead. On most MS-DOS disks, 93% of the formatted media is available for data blocks.

4.6. MS-DOS Limitations

MS-DOS is primarily a file system intended for use on a single-user microcomputer. There are many significant limitations to the MS-DOS file system in a multiuser or networked environment.

The root directory is a fixed size. Even though a generous number of root entries exist, once the entries are consumed, no files can be created in the root directory. Also, disks no larger than 128 megabytes can be supported.

There are many serious limitations to MS-DOS's file attributes. Files may not have holes. Files may not be of zero length and can have an end of file character. There are no file identification numbers, nor is there any ownership information. File permissions are minimal. No hard or soft links or any special devices can be created or accessed in an MS-DOS file system. Finally, there is no method of storing a file generation number within the directory entry.

5. THE SYSTEM V PROTOTYPE

In one version or another, the System V file system is used by the V6, V7, System III, System V, XENIX⁸, 4BSD, and 4.1BSD operating systems. More operating systems provide access to System V file systems than for any of the other file system prototypes.

System V's file system supports file system block sizes in units of 512 bytes, 1K, and 2K (only for A. T. & T.'s 3B5 line of computers). File names can contain up to 14 characters. All ASCII characters except '\0' and '/' may be used within file names. Also the file names '.' and '..' are reserved. Most implementations of the System V file system do not permit the high order bit set within each character.

⁸ XENIX is a trademark of Microsoft Corporation

5.1. System V Architecture

A System V file system contains four components: a 512 byte boot block, a 512 byte super block, the on-disk inode table, and file system data blocks.

5.1.1. System V Super Block

The System V super block contains data describing the file system. The principal components are: the size of the on-disk inode table, the size of the file system, a count of free data blocks on the disk including a list of 50 free blocks, a count of free on-disk inodes including a list of 100 free inodes, and identifiers to determine the validity and block size of a file system.

5.1.2. System V Directory Description

System V directories are constructed from not necessarily contiguous blocks containing fixed length directory entries. Each directory entry is 16 bytes long and is defined as follows:

```
struct sysv_dir {
    u_short    sysv_ino;
    char       sysv_name[14];
};
```

The `sysv_ino` field contains a unique file identifier number (the inode number). Having this inode number allows for easy retrieval of the file attributes structure (the on-disk inode).

5.1.3. System V On-Disk Inodes

With the exception of the file name and the inode number, the on-disk inode contains all the data about the file. Each of these structures is 64 bytes long and is defined as follows:

```
struct sysv_inode {
    u_short    sysv_mode;
    short      sysv_nlink;
    u_short    sysv_uid;
    u_short    sysv_gid;
    int        sysv_size;
    u_char     sysv_addr[40];
    u_int      sysv_atime;
    u_int      sysv_mtime;
    u_int      sysv_ctime;
};
```

The `sysv_mode` field contains both protection information and a description of the file type. The protection stored is typical for a UNIX system, three types of protection (permit read, write, or execution) for three levels (owner, group member, or world). The remaining fields in `sysv_mode` describe the file type (for example, block or character special device), and execution attributes (for example, set user ID on execution).

The `sysv_nlink` field allows a number of file names to resolve to the same file. The `sysv_uid` and `sysv_gid` fields describe the owner. The `sysv_size` field gives the byte size of the file. `sysv_atime`, `sysv_mtime`, and `sysv_ctime` hold the last file access time, the last file modification time, and the last inode change time.

The `sysv_addr` field provides a VBN to LBN mapping for the file. The 40 bytes hold thirteen three-byte disk block numbers. The first ten disk blocks numbers are the direct blocks of a file. The eleventh entry is the first indirect block. This indirect block contains a list of up to 128 blocks that are attached to the file. The twelfth entry is the second indirect block. The second indirect block contains a list of up to 128 blocks that are themselves indirect blocks. The last

entry is the third indirect block. It adds another level of indirect blocks.

5.2. System V Prototype Design and Implementation

The System V file system prototype was based loosely on the System V Version 2 Release 2 (V.2.2) source tape. During implementation, much code was taken and reused from the existing UFS code. The only source that was taken from V.2.2 was the block and inode allocation and deallocation code. Most of the path name translation (*sysv_namei*) and file I/O (for example, *sysv_rwgp*) code was taken directly from the UFS prototype.

5.2.1. System V Block and Inode Allocation

Much like the block map within the *sysv_inode*, free blocks and inodes are stored in an array. The last entry in the free block (and inode) array points to a block containing 50 more indices to free blocks. Likewise, the last entry in this block points to a block containing 50 more free block indices.

The System V *mkfs* command understands disk geometries and constructs the free list using an optimal block layout.

Unfortunately, allocating and deallocating blocks occurs frequently. This causes the list to lose its optimal ordering. Since the allocation code simply removes the next block off the list, a file's data blocks can become scattered over the file system.

5.2.2. System V Prototype Functionality

The System V operating system does not support all of the file system related system calls found in 4.3BSD. The GFS System V prototype however, provides code for making and removing directories, renaming a file, truncating a file to a specific, potentially non-zero length, and insuring all cached data blocks are flushed back to disk. With the exception of symbolic links, all file system functionality found in 4.3BSD was implemented in the System V prototype.⁹

5.3. System V Performance

Read and write performance tests were done to a newly created System V file system. This file system was created with file system block size of 1K. Since the free block array was optimally ordered, disk head movement was minimal. This provided for "best case" I/O measurements. Measured performance follows:

I/O Performance (in kB/sec)			
	512b	1K	8K
Write	47	47	48
Read	35	37	38

As with the MS-DOS file system, the file system bandwidth is limited by the small transfer size.

5.4. System V Support for NFS

The on-disk inode for the System V file system contains no space to hold a file generation number. Therefore removing a file and reusing its on-disk inode can cause inconsistencies in NFS. Symbolic links are not supported. Attempting this functionality fails in an NFS environment. Finally, the small file system block size causes a degradation in NFS I/O performance.

⁹ There is space in the *sysv_mode* field of the on-disk inode to identify a symbolic link. This creates a file system that is not transportable to a System V operating system.

5.5. System V Strengths

The System V file system is easily understood. Users not familiar with file system implementations can fix corruption with a high probability of success. The file system also supports most of the 4.3BSD file system semantics.

5.6. System V Limitations

The System V file system uses a small file system block size. The small block size increases the number of transactions a device must handle and reduces its effective throughput. Experience has shown that larger machines demand considerable I/O bandwidth. I believe that the current System V file system will be inadequate on a large machine. If the file system block size was to increase (as in the file system for the 3B5), disk space would be wasted on partially filled data blocks.

The functionality for symbolic links cannot be supported without causing problems when moving the file system to a System V machine. NFS encounters difficulties since file generation numbers cannot be stored. Finally, the `sysv_size` field in the System V inode limits the file size to 2^{31} bytes.

6. THE UFS PROTOTYPE

UFS, or the fast file system from 4.3BSD (McKus1983a), evolved from the System V file system. Major improvements were made to increase reliability and performance.

UFS supports file system block sizes of 4K and 8K. These blocks may optionally be broken in 2, 4, or 8 pieces (fragments). File names can contain up to 255 characters. UFS permits the same character set within names as does the System V file system (everything but the names ':' and '.' and the characters '\0' and '/' are allowed).

6.1. UFS Architecture

A UFS disk contains two components: an 8K boot block, and many cylinder groups. These cylinder groups permit blocks to be allocated while attempting to minimize disk head movement.

6.1.1. UFS Cylinder Groups

A cylinder group consists of five components: a super block (or copy thereof), a cylinder group structure, some on-disk inodes, a cylinder group summary structure, and some data blocks. These cylinder groups are spread over the entire file system. This differs from the other prototype file systems in that UFS scatters file system data structures across the surface in an attempt to minimize disk head movement.

The UFS super block contains a description of the file system and the media. Data stored there includes the geometry of the disk, the file system block and fragment size, the size of a cylinder group, and file system configuration parameters.

A cylinder group structure contains the size of the cylinder group, the location of last used on-disk inodes and blocks, and a used on-disk inode map. Also included in this structure is a free disk block map for the cylinder group. The cylinder group summary structure holds the summary of available resources within the group.

6.1.2. UFS Directory Description

UFS directories are constructed from not necessarily contiguous blocks containing variable length directory entries. Each directory entry is potentially 264 bytes long and is defined as follows:

```

struct ufs_dir {
    u_long    ufs_ino;
    u_short   ufs_reclen;
    u_short   ufs_namelen;
    char      ufs_name[256];
};

```

`ufs_ino` uniquely identifies a file as does `sysv_ino` in the System V on-disk inode. The `ufs_reclen` field describes the length of the UFS directory entry. The `ufs_namelen` field contains the length of the file name. The name length is always in multiples of 4 preventing memory addressing problems. Each of these entries are wholly contained within a disk block and are stored as compactly as possible within a directory.

6.1.3. UFS On-Disk Inodes

As with the System V file system, all information other than the file name and file identifier (on-disk inode number) is held in the inode. The UFS inode structure is as follows:

```

struct ufs_inode {
    u_short   ufs_mode;
    short     ufs_nlink;
    short     ufs_uid;
    short     ufs_gid;
    quad      ufs_size;
    timeval   ufs_atime;
    timeval   ufs_mtime;
    timeval   ufs_ctime;
    long      ufs_db[12];
    long      ufs_ib[3];
    long      ufs_blocks;
    long      ufs_gennum;
};

```

Differing from the System V file system is `ufs_size`, `ufs_atime`, `ufs_mtime`, and `ufs_ctime` which are 64 bits long. These times are in microsecond resolution (depending on the resolution of the machine's clock). Also `ufs_blocks`, the number of blocks allocated to the file, and `ufs_gennum`, or file generation number, have been added to the basic System V on-disk inode.

6.2. UFS Prototype Design and Implementation

The UFS prototype is based strictly on ULTRIX 2.0 UFS code. In fact, it has been the basis for much of the MS-DOS, ODS-II, and System V prototype file system code.

6.2.1. UFS Block and Inode Allocation

The UFS disk block allocation policy attempts to allocate all data blocks for a file in the same cylinder group. Further, the policy attempts to position each block in a rotationally optimal position relative to the position of the previous block in the file.

If the optimal block has been previously allocated, the allocation code first attempts to allocate a block within the same cylinder group. If there are no free blocks in the cylinder group, the code quadratically searches other cylinder groups. If an available block still has not been located, a brute force search is conducted.

For a file, the inode allocation strategy attempts to place the file within the same cylinder group as the parent. When allocating an inode for a directory, the allocation is done from the cylinder group that has the fewest allocated inodes.

6.3. UFS Performance

As expected, the increase in file system block size and the block allocation policy allows for an increased I/O bandwidth. Read and write performance for an 8K block size, 1K fragment size follows:

I/O Performance (in kB/sec)			
	512b	1K	8K
Write	85	148	176
Read	118	154	206

6.4. UFS Support for NFS

As distributed by UCB, the 4.3 UFS on-disk inode does not hold a file generation number. Since unallocated space exists within each on-disk inode, a generation number is held. This is the only functionality that was needed for support of NFS.

6.5. UFS Strengths

Having a large file system block size reduces the number of transactions disks must service. Since UFS obtains data about disk geometry, the placement of newly allocated blocks and inodes is close to optimal. These improvements markedly increased file system throughput.

By allowing large disk blocks broken into fragments, disk space is also put to good use. Finally, by definition, the file system does an excellent job of supporting UNIX file system semantics.

6.6. UFS Limitations

The complicated allocation scheme can become a burden for slower CPUs. Also, corruption cannot be corrected as easily as in the System V or MS-DOS file systems.

7. LIMITATIONS OF THE GFS INTERFACE AND THE UNIX SYSTEM

After prototyping the ODS-II, MS-DOS, System V, and UFS file systems, the limitations of GFS and the UNIX system are better understood.

The file system buffers in GFS can be no larger than 8K without restructuring the buffer allocation strategy. Unfortunately, the change is not simply increasing this 8K limitation. System page table sizes should be adjusted, and the buffer allocation code should be changed so as to not limit the buffer size.

The UNIX system call interface provides no method to instruct file system implementations to alter their block allocation strategy. ODS-II permits contiguous files, but the system call interface provides no mechanism for instructing the implementation to do so.

UNIX file protections are an issue. At one end of the spectrum, MS-DOS has limited protection and the file system becomes unusable on a non-friendly machine. At the other end of the spectrum, ODS-II has four permission modes that can function on four different levels. While most of these permissions can be handled, ODS-II's ACLs still cannot be addressed.

Finally, many parameters to file system related system calls assume a 32 bit quantity but files in both UFS and ODS-II can be larger than 2^{32} . The system calls *lseek*, *read*, *write*, and *truncate* all take a 32 bit parameter specifying a length or offset.

8. CONCLUSION

We have learned many lessons from prototyping the ODS-II, MS-DOS, System V, and UFS file systems. The GFS interface is better understood; any needed functionality has been added. We know how NFS functions when serving each of the file systems and have measured basic file system performance using several different file system architectures.

Much work still needs to be done to UNIX and GFS. The UNIX system calls and GFS interfaces using a 32 bit offset or size must be changed to support very large files. Structures holding block indices must be changed to support very large media. Support must also be added for write once media. Finally, the buffer caching code needs to be restructured.

While the work done for this paper is strictly research, the information presented should help file system performance in the future.

9. ACKNOWLEDGEMENTS

Bob Rodriguez deserves special mention. He was the author of the ODS-II prototype that was the basis for the ODS-II/GFS file system.

My management, Steve Reilly, Fred Glover, Dave Cardos, Kent Ferson, Glenn Johnson, Bill Heffner, Jack Smith, and Ken Olsen have graciously allowed me enough time to design and implement the file systems.

Discussions on the GFS interface with Mike Karels and Kirk McKusick from the University of California at Berkeley were timely. Jim McGinness, Dave Roberts, Ken Reilly, Ricky and Larry Palmer, Chet Juszczak, and Jeff Chase answered questions and offered much needed insight.

Finally, I thank the cast of people that critiqued this document.

10. REFERENCES

Readers will find a more complete reference to specifics on file systems and their implementations in the following works:

References

Rodri1986a.

R. Rodriguez, M. Koehler, R. Hyde, "The Generic File System," *USENIX Conference Proceedings*, pp. 260-269, Summer, 1986.

McKus1983a.

M. K. McKusick, W. Joy, S. Leffler, R. Fabry, "A Fast File System for UNIX," *CSRG Technical Report*, vol. 83, no. 147, 1983.

Golds1983a.

A. Goldstein, Files-11 On-Disk Structure Specification, 1 Oct, 1983.

Norto1985a.

P. Norton, *The Peter Norton Guide to Programming the IBM PC*, pp. 99-125, Microsoft Press, 1985.

Ritch1978a.

D. Ritchie, K. Thompson, "The UNIX Time-Sharing System," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1905-1929, July - August, 1978.

Sandb1985a.

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings*, pp. 119-130, Summer, 1985.

Thomp1978a.

K. Thompson, "UNIX Implementation," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931-1946, July - August, 1978.

Now UNIX¹ Talks To Me In My Language

*Pascal BEYLS
BULL
1, rue de Provence
98432 Echirolles
FRANCE
..!mcvaz!inria!echbull!xopen*

ABSTRACT

BULL and SIEMENS, 2 major European companies, have jointly achieved the internationalization of UNIX, as defined by the X/OPEN² group. This document describes a part of this subject, which is called "Message Presentation".

Today, in order for applications to be accepted by users in other countries, they must present a user interface in the user's native language.

An original solution, based on a new section in the **COFF** (Common Object File Format), has made it possible to eliminate any multilingual problems, so that different users may now work in different languages on the same machine and at the same time.

It is noteworthy that it is two European, non-English companies which are offering a truly European, if not international, UNIX.

1. Unix is registered trademark of AT&T in the USA and other countries.

2. X/OPEN is a licensed trademark of the X/OPEN Group Members.

1. Introduction

1.1 Generalities

The internationalization of UNIX has been achieved by doing work in three distinct areas:

- allowing users to use new character sets. The ASCII character set is unacceptable in a language environment other than English, due to the number of accented characters and other symbols (the new ISO 8859 standard contains all the letters and symbols necessary used in Western European languages).
- allowing for differences in the different cultures (date formats and money symbols are two examples)
- allowing users to "talk" to the computer in their own language.

The points listed above have been standardized by the X/OPEN group. This document will concentrate on the last point : Message Presentation.

Message Presentation is a way to allow programs to interact with users in different languages. In the past, when a program was to be exported to a country with a different language than that used in the original program, the entire source program had to be re-read and all the messages translated into the new language. There are several disadvantages to this method:

- one has to have the program source in order to translate the messages.
- the new messages are hard coded into the program source. There must be one copy of the source for each language.
- once the program is translated, the entire program has to be re-compiled.
- each translated program becomes a new version of the program, and has to be maintained, which complicates the job of support personnel.
- since the internationalization has changed the source, you have to test the program to make sure the program logic has not been accidentally changed.

1.2 Basic Requirements

There are several conditions that must be met in a serious solution for Message Presentation:

- The programmer must be able to program in his native language without having to worry about language problems.
- It should not change the way the programmer does his job.
- Translation of the program for different countries must be possible without using the program source. This allows you to have only ONE version of the program sources, not one for every language. Errors added during translation of the source file are thus avoided.
- The same program on the same machine should be able to talk to several users in different languages at the same time. The language is made available to the program by an environment variable LANG.

The last condition is imperative. European organizations such as the EEC (European Economic Community) need such flexibility, as their user community may speak any one of a dozen different languages.

Two types of message presentation are possible.

- Dynamic message presentation:
this is the method described above.
- Static message presentation:
the program (executable) can be "frozen" to speak only one language. This actually covers most of the cases, as most customers only want to talk to the machine in their native language and do not have not multilingual staff.

A static message presentation can be much more economical in terms of memory usage, disk usage and CPU time. Dynamic message presentation is absolutely necessary in some cases, but static presentation has several enticing advantages. Both should be available so as to give developers and users the flexibility to choose the best solution for their needs.

1.3 The choices

The constraints listed above eliminate the archaic solution of putting the messages in the source file. Several possibilities were considered, here we will concentrate on two of them:

1. a message catalog
2. modifying the COFF

1.4 The message catalog approach

Every message in the source program is located and is replaced by a function call:

getmsg(fd, msg_num)

where **fd** is a file descriptor indicating the file where the messages are stored, and **msg_num** is the number of the associated message. This solution consists of replacing a char pointer with a call to a function that returns a pointer to the "translated" string. The messages associated with the programs are contained in a separate file. Tools can be made to help with the automatic extraction of message text and its replacement with calls to the proper function(s).

This method has its drawbacks, however:

- Initialized static variables and global variables can not be replaced by calls to a function. These types of strings often represent 30 or 40% of the messages in a program.
- Substitution of pointers by function calls engenders an overhead, namely an extra file descriptor and the time to read the messages from the file.
- The sending of a program (by uucp, for example) would also mean the sending of a message catalog file for each language that the program should be able to speak. Without a message catalog, the program is worthless.
- The program will only be usable when the message catalog file is available. If it is located on a different mountable volume than the program, the program depends on two file systems, not one. A "cleanup" of the file system where the message catalog is located effectively inhibits usage of the program, even though the program is still available.
- The message catalog approach is not adapted for use with .o or .a files (libraries and archives). A separate operation is thus necessary during the link editing phase.
- Problems arise when trying to access the message catalog. How to distinguish the message catalog files for programs with the same name (a.out, for example....).

All of these constraints, especially the first, do not allow us to realistically consider the first possibility as a viable solution. A solution that eliminates these problems has been found, and is based on an extension to the COFF (Common Object File Format).

1.5 The COFF-based Solution

It was decided to incorporate the messages in a new section of the COFF. The COFF was designed flexibly enough to allow for definition of new sections, and lends itself perfectly to what we want to do. It was also decided to do all the extraction of messages by the compiler. The basic idea is analogous to that used in `xstr(1)`. This frees the programmer from all the busy-work that is easily done by a program and allows him to concentrate on programming.

file header
optional information
header section ".TEXT"
header section ".DATA"
header section ".BSS"
raw data ".TEXT"
raw data ".DATA"
reloc info ".TEXT"
reloc info ".DATA"
line numbers ".TEXT"
line numbers ".DATA"
SYMBOL TABLE
STRING TABLE

Figure 1
Present structure of a ".o" or a ".out"

2. The COFF solution

The basic requirement is that the mechanism should not modify the way a programmer writes programs, nor its associated makefiles. It should limit the amount of extra work necessary for a programmer to make a multi-lingual program.

2.1 Basic principles

The mechanism of message presentation is integrated with the development tools: *cc*(1), *as*(1), and *ld*(1). It is made up of:

- an evolution of some of their constituent parts
- a set of pre/post processors inserted into the development chain.

The mechanism of internationalization is invoked as an option to *cc*. The programmer does not need to manipulate an intermediary work file.

The message presentation system has the following basic principles:

- A new section in the COFF is defined to hold the messages separate from program logic. All the messages in the program in the same language are grouped in the same section as defined in an extended COFF format. The message section(s) is (are) included in the executable (*a.out*) file. The new section has type **message** and is identified by a new flag **STYP_NL** in the section header (see *a.out*(4)).
- An extension to the loader *exec*(2) reads into memory only the message section associated with the user's declared language (environment variable **LANG**).
- There is a translation tool that helps the programmer (or a professional translator) associate the program's messages with messages in other languages, to facilitate the translation into multiple languages, without modifying the program source.

2.2 Important criteria

- the new *exec*(2) must be able to load programs in both COFF formats (the old format and the new international one)
- the message presentation mechanism must work properly for applications that are developed using separate compilation.
- libraries (*libc*, *libm*, user libraries ...) must also be able to hold multi-lingual messages so that the same library can be used in many different language environments. Thus :
 - the *.o* files must be translatable and be able to contain several different message sections, one for each language.
 - the link editor (linker) must be able to link these multi-lingual *.o* files, correctly combining the proper language sections with each other.
- the message presentation mechanism collects both printable and non-printable strings (which should NOT be translated) in the source. We plan on having such strings markable by the programmer (in a library or *.out*) so that they will thus not be translated.
- performance
 - during program development (performance is not very important)
 - during program loading (into memory)
 - during program execution
- the recognition of a particular language by the linker is done using an eight-letter (maximum) string.

We must also take into account the following points:

- portability, especially vis-a-vis
 - other compilers
 - other message presentation systems
 - other operating systems.
- allowing text segments to be shared by several users at once.
- compatibility with shared libraries.

3. The new development cycle

3.1 Generalities

In **C**, a string of characters is always manipulated as an address of the string located in the *.data* section. This address is hard coded into the *.text* section.

The idea is to isolate the strings and to put them into a new section of the **COFF** called the "message section". This "message section" (which has the name *.nl* for *native language*) contains all the messages of the program in any given language. In order to "translate" the program into a new language, all that need be done is to extract the "message section" from the **COFF**, translate the messages, and then re-insert the new messages into a new (separate) *message section* for the new language. All the messages are in the same *a.out*, each language having its own separate section, and all the messages of a given language are in the same section.

But, the messages in different languages are normally of different lengths than the original. The addresses of the individual messages thus varies from language to language. Since the addresses are directly written into the *.text* section, it becomes necessary to modify the program text for every execution to change the addresses, and this during loading! Not a very practical thing to do.

To overcome this problem, the addresses of the character strings are put into an array of pointers, called *_nl_st*. We are thus no longer obliged to change the address of a string mixed in with the program instructions, but manipulate elements of an array, which are the addresses of the strings in question. Since this array's contents change for each execution, it will be initialized at load time. Accessing a string is now done by double indirection.

3.2 Solution

In order to reduce the modifications necessary to the existing program development cycle (*cpp*, *ccom*, *as*, *ld*), these tools are complemented by independent pre/post processors:

- a compilation pre-processor : *nl_cpp*
- an assembler pre-processor : *nl_as*
- a linker post-processor : *nl_ld*.

Linking has to be done in two passes in order to allow the possibility of static presentation of messages. Between the two passes, the "*a.out*" file is modified by *nl_ld*.

These pre-processors are called optionally from the *cc* command. Thus one can still use the "old" set of development tools. The relationship of the different parts of the international development tools is:

```
cpp → nl_cpp → ccom → nl_as → as → ld -r → nl_ld → ld → translator
```

As well as introducing the new processors *nl_cpp*, *nl_as* and *l_ld*, *as* and *ld* must undergo slight modification.

4. The *nl_cpp* pre-processor

In a **C** program, we want to find all the strings in order to easily print out their translation in any given language during execution. We must be able to change the references to a string without having to change the *.text* section, only the *.data* section. The *nl_cpp* pre-processor modifies the **C** program source in order to isolate all the strings in a module and to generate indirection when it does not already exist.

A string in **C** can show up in one of three ways:

1. `char *p = "abc";`
2. `char t[10] = "efg";`
3. `f(...,"ggg",...);`

Declarations **1.** and **2.** can either be outside of a function (global variables) or inside a function (local variables). In the second case, if the variables are local to a function, they must be declared static (restriction imposed by the **C** language).

4.1 Source transformation

- Declaration at the beginning of the source file of an array of pointers to strings : **static char *_nl_st[...]**. This array is internal to the module (static) and contains as many entries as literal strings found in the source module. This table is **not** initialized.
- Modification of the original initialized string declarations (cases **1** and **2** above):
 - global declarations:
 1. `char *p = "abc" + x;` becomes `char *p=x;`
 2. `char t[10] = "efg";` becomes `char *t;`

Notes:

Declarations as in **2.** above are not recommended in multi-lingual programs for these reasons:

- a. the size of the string is fixed and thus one must verify during translation that the maximum size defined by the programmer has not been exceeded (10 characters in the preceding example).
- b. All the external references to this array, such as:
extern char t[];
must be manually replaced by
extern char *t;

nl_cpp should output a "warning" if it finds declarations of type **2.**

— local declarations (inside a function) :

- a. automatic variables:
`char *p = "abc";` becomes `char *p=_nl_st[i];`
- b. static variables:
`static char *p = "abc";`

The pointer *p* in this case is unknown outside of the function and thus cannot be modified outside of the function. We cannot initialize it either, as we could an automatic array, as it is illegal to initialize a static variable

with the contents of another variable. And this initialization would happen for every call to the function, which is not what we want (the value of static variables should not change between calls to the function).

We introduce initialization instructions:

```
static char *p = 0;
static short _nl_inista = 1;
if(!_nl_inista)
{
    p = _nl_st[i];
    _nl_inista = 0;
}
```

- Modification of references to constant strings:

`f(...,"bbb",...);` becomes `f(...,_nl_st[j],...);`

- Declaration of three elements at the module's end:

1. a structure `_mg_hd` (called the group header) containing general information, useful during translation:
 - the file name: `mg_name`
 - the number of extracted messages: `mg_mcnt`
 - the address of the array `_nl_st`: `mg_data`
 - two fields eventually initialized during translation: `mg_size` and `mg_offs`
2. an array of `_msg_hd` structures (message header) containing specific information about each message:
 - a flag indicating if the initialized variable is an array or not: `msg_flag`
 - the pointer address to initialize at load time : `msg_init`
 - the length of the message: `msg_leng`
 - two fields eventually initialized by the translator: `msg_offs` and `name`
3. an array of pointers to messages (strings), identical to the array `_nl_st` but initialized.

4.2 Example of source transformation

initial program

```
char t[10] = "abc";
char *p = "efgh" - 1;
f()
{
    static char *p = "ijk";
    char *p2 = "lmn";

    g("opq");
}
```

program output by nl_cpp

```
# 1 "es.c"
static char *_nl_st[5];
# 1 "es.c"
char *t = 0;
char *p = 0+1;
f()
{
    static char *p = 0;
    char *p2 = _nl_st[3];
    # line 7 inserted by nl_cpp in "es.c"
    static short _nl_inista=1;
    if (_nl_inista)
    {
        p = _nl_st[2]; _nl_inista=0;
    }
    # 7 "es.c"
    g(_nl_st[4]);
}
```

At the end of the program,
we find the following structure declarations

```
static struct
{
    char mg_name[14];
    unsigned short mg_mcnt;
    char **mg_data;
    long mg_size;
    long mg_offs;
} _mg_hd = {"es.c",5,_nl_st,0,0};
```

```
static struct
{
    unsigned short msg_flag;
    char **msg_init;
    unsigned short msg_leng;
    long msg_offs;
    char name[8];
} _msg_hd[5] = { {0x04,&t,10,0," "},
                {0,&p,5,0," "},
                {0,0,4,0," "},
                {0,0,4,0," "},
                {0,0,4,0," "}};
```

```
static char *_msg[5] =
{
    "abc\0\0\0\0\0\0\0",
    "efgh",
    "ijk",
    "lmn",
    "opq",
};
```

4.2.1 General comments

- In the preceding example, the 0th entry in the array `_msg` corresponds to the array `t` in the original program source. In order to reserve 10 characters, `nl_cpp` generates the string `abc\0\0\0\0\0\0\0\0\0\0`.
- the structures `_mg_hd` and `_msg_hd`, as well as the array of pointer to messages `_msg` are declared at the end of the `C` source module, and as such the assembler directives concerning their reservation will be at the end of the assembler source, just before the string reservations.
- The "`a.out`" that comes out of the internationalized production chain is no longer directly executable because the pointers `p` et `t`, as well as the array `_nl_st` are not initialized. An updating of the "`.data`" section will be done after loading the program, using the addresses `mg_data` et `msg_init`.

5. The `nl_as` preprocessor

This pre-processor modifies the assembler source in order to generate two structures, `_mg_hd` and `_msg_hd`, the array of pointers to messages `_msg` and the message strings into the message section "`.nl`". It transforms all the "section 15" directives into "section `.nl`" directives, working from the `_mg_hd` structure declaration.

6. Assembler changes

Since `as` only knows how to deal with `.text`, `.data`, and `.bss` sections, it must be modified to recognize the new `.nl` section.

7. Binary format (before linking)

The "`.o`" files made by the assembler will now have four sections: "`.text`", "`.data`", "`.nl`" and "`.bss`". Some of these modules could be translated before the link phase (as would be the case for libraries). This translation would create other message sections in the file, one section for each added language.

Following is a diagram of the placement of messages in the "`.nl`" section, and the links between the group and message headers, and the data (see figure 2).

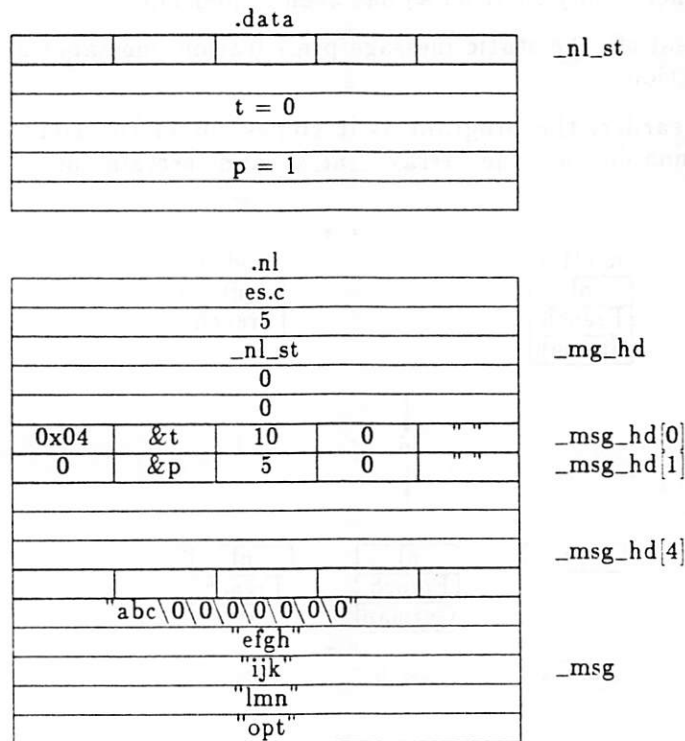


Figure 2

8. Linking

8.1 a.out format

1. *ld* normally knows how to treat a certain number of sections. By default, it places the *.text* and *.data* sections at pre-determined addresses (the *.bss* section is added to the end of the *.data* section) and inserts the other sections in unused addresses in the virtual memory space. The loading mechanism and the execution of an *a.out* demand that the *.data* and *.bss* sections be placed at the end of the virtual address space. The message sections will thus be put between the *.text* and *.data* sections. So as not to waste space in the virtual address space, and because only one of the (possibly) many different message sections is loaded by *exec*, the message sections in *a.out* all have the same virtual address.
2. The *.o* from an application can be translated (i.e. several versions of every string, each in a different language) before the link phase. It can thus contain several message sections, and there are three possibilities for the *a.out* after the link editing phase (see figure 3):
 - a. *ld* only keeps the message sections for one (specified) language. The linking is refused if any module does not contain that language's module.
(→ rather restrictive).
 - b. *ld* keeps the message sections found in the *.o* files.
(→ the *a.out* will not be executable in every language)
 - c. *ld* keeps only the message sections that are defined in all of the modules.
(→ some translations will be lost)

For the moment, only solution **a)** has been implemented.

3. In order to satisfy the static message presentation, messages are put at the end of the `.data` section.
4. As we saw earlier, the program as it comes out of the compilation phase is not directly runnable, as the array `_nl_st` and certain pointers are no longer initialized.

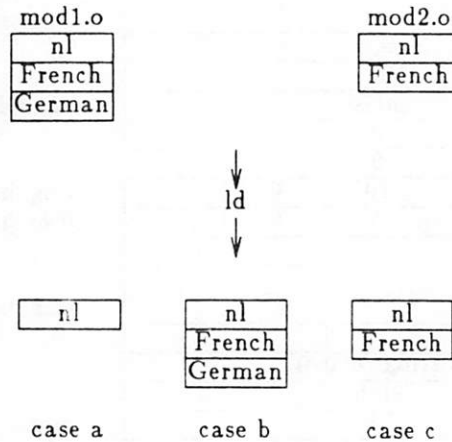


Figure 3

8.2 *ld* modifications

1. Putting message sections between the `.text` and `.data` sections can be implemented by giving directives to *ld*. We create a new type of section: the *message section*.
2. Putting messages at the end of the `.data` section means that we must keep relocation information for symbols in the `.bss` section. The linking is now done in two passes, and *nl_ld* modifies the *a.out* between the two. *ld* is first called with the `-r` option, which preserves relocation information.
3. The messages placed after the `.data` implies having to translate the references into the `.bss` section. The symbol relocation information in the `.bss` section must therefore be saved. The solution is that the linking be done in 2 passes between which the processor *nl_ld* modifies the "*a.out*". The first linking is done with the `-r` option of *ld*.
4. A new startup routine, *nl_crt0.o* which initializes the array `_nl_st` (containing the pointers to the messages), is linked into the program.

9. The *nl_ld* post-processor

The post-processor *nl_ld* modifies the *a.out* file between the two passes of *ld*, to allow a static presentation of messages (the "freezing" of the program in one language). The resulting file (after running through the second pass of *ld*) can still be used for dynamic message presentation, but all the "hooks" are there for later transformation of the *a.out* if one wishes to do so. In order to "freeze" the program in one language, all the messages in the *message section* must be put back in their initial place, at the end of the `.data` section. This means that the `.bss` section will be moved to a new place and all the references to it must be modified. *nl_ld* makes available certain information to allow this. Using the relocation information in the *a.out* left by the first pass of the linker (*ld -r*), it builds a new section `.reloc` that contains all the relocation information

for the symbols in `.bss`.

Also, in order to speed up the loading and startup of the program, which entails finding the message section corresponding to the user's language and initializing the array `_nl_st` (which contains the pointers to the messages) with the correct addresses, `nl_ld` adds a header to the `".nl"` section that allows it to find strings more quickly. In order for this to work, the `.nl` section must be complete; all the library functions (especially `nl_crt0`) must have been linked in during the first pass of the linker.

`nl_ld` adds following information :

- the `_md_hd` structure, which contains:
 - the size of the section
 - the number of message groups contained in the section
 - the total number of messages
- an array of pointers to the `_mg_hd` structures

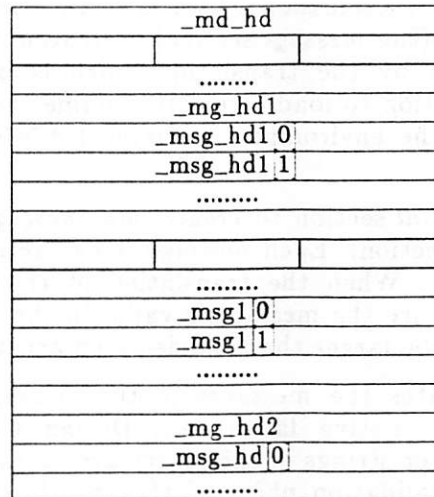


Figure 4
".nl" section of COFF

10. Loading an executable file

The solution consists of:

- modifying `exec(2)` so that it only loads into memory the message section corresponding to the `LANG` environment variable.
 - the kernel does not usually know anything about the user environment; we have added a field (`u_lang`) to the user structure inherited from the parent. During loading, if `exec` finds message type sections in the executable file, it uses the `u_lang` field to decide which message section to load.
 - `char u_lang[8]` can be changed by a system call (`setlang(2)` and `getlang(2)`).
 - the kernel might not know the process causing an error; the error message should be printed on the system console in the system's language (the system administrator only knows one language)

- `char u_lang[8]` can be changed by a system call (`setlang(2)` and `getlang(2)`).
- the kernel might not know the process causing an error; the error message should be printed on the system console in the system's language (the system administrator only knows one language)
- `exec(2)` recognizes each section by the name written at the beginning of each section: `char s_name[8];`
- putting the message section between the `.text` and `.data+.bss` segments
 - the sharing of messages between different processes is a possibility.
 - the different message sections should all have the same virtual address.
- arranging for address initialization of strings contained in the data segment by the runtime routine `nl_crt0` (updated according to the information contained in the message section). This introduces a slowing down of the loading process, due to the time needed to initialize the array of pointers to the messages (`_nl_st`).

12. The translation tool

The translation tool is a tool for a professional translator to translate the program into other languages. A new section (the *message* section) is created in the *a.out*. The name given to this section is chosen by the translator, which is used during loading to determine if it is the proper section to load or not (the name should have a relation to the language, as the value of the environment variable `LANG` is used to determine which *message* section to use).

The translator works from the `.nl` section to create new *message* sections, which have the same structure as the `.nl` section. Each message is extracted from the `.nl` section and proposed to the translator. When the translation of the message is given, the translator tool checks to make sure the message is valid (in the case of globally defined arrays, the new message cannot be larger than the declared array size).

The translator tool also validates the messages in the *message* section `".nl"`, which contains the messages in the "native language". During the message extraction performed by `nl_cpp`, all character strings in the source are extracted, and some are not meant to be translated. The validation phase of the translator is used to mark the strings (or character arrays) that are not meant to be translated.

12.1 Validation of the `.nl` section

The validation of messages must be done by the programmer. Marking a string as "non-translatable" makes it invisible to the translator, who shouldn't translate it. A flag is set in the header of the message section that says that a particular message shouldn't be translated. That message will no longer appear in future translation sessions.

12.2 Message translation

Message translation can be seen from two different viewpoints:

— interactively:

The translation is done interactively with a screen-based translation "editor" (based on *curses*). The translation isn't necessarily done all at once, and can be stopped and started again later. The verification of a message is done when the translation for the string is entered. A *message* section is added to the `COFF`, so as to not lose the translations already made, even though all the messages haven't been translated yet. This partially-finished section will be extracted and work will continue at the same spot during the next session.

— from a file:

The user can give a file containing the translations as input to the translator. After checking the validity of the messages, the new *message* section corresponding to the language given by the translator is created.

In all cases, a new *message* section is added after the last one already in the file and its file address is calculated so that it will have the same virtual address as the *.nl* section.

12.3 Other services

- the possibility to erase a message section associated with any given language.
- listing the languages of messages in the binary (similar to *dump -h*).
- extraction of the message section of a given language (useful if one wants to re-do the translation).
- transformation of an internationalized *a.out* into an "normal" *a.out*. This "normal" *a.out* is made from the *.bss* relocation table left by the linker. The transformation tools install the messages at the end of the data section and resolves the references into the *.bss*. This tool does a lot of the same things as *ld(1)*, and is fairly easy to make.
- examine the problem of the maintenance and the updating of messages (creation of a library file containing the messages in the original language and all the translations done so far).

file header
optionnal informations
header section ".text"
header section ".nl"
header section ".data"
header section ".bss"
header section "reloc"
header section "FRAN"
header section "ENG"
...
header section "ITAL"
raw data ".text"
raw data ".nl"
raw data ".data"
raw data "reloc"
raw data "FRAN"
raw data "ENG"
...
raw data "ITAL"
reloc info
SYMBOL TABLE
STRING TABLE

STRUCT OF AN INTERNATIONALIZED PROGRAM

A UNIX[®] System V STREAMS TTY Implementation for Multiple Language Processing

Hiromichi Kogure

Richard McGowan

AT&T Unix Pacific Co., Ltd.

2-21-2 Nishi-Shinbashi,

Minato-ku, Tokyo, Japan

ABSTRACT

The release in May 1987 of the Japanese Application Environment, Release 2.0 (JAE 2.0) marks the first release of a STREAMS-based tty subsystem under UNIX^{*} System V. The STREAMS architecture provides for a flexible full-duplex connection between device drivers and user processes. It allows for great flexibility in processing multiple languages, since modules can be developed for specific purposes without affecting the whole of the tty subsystem. Switching human languages requires changing only those modules of the subsystem specifically related to that language. Specific examples of the new architecture are drawn from the implementation of an input system supporting the Japanese language. The international aspects of this work revolve around AT&T's Supplementary Code Sets for UNIX System V (SCS), a method of codeset mapping which provides a firm foundation for processing many different languages. The generality of this mapping system and its applicability to various languages is discussed. Strategies used to program in a general way for use with multiple languages and SCS are illustrated. Finally, there is some discussion of the types of development tools and user-level tools used with the STREAMS tty subsystem.

1. Introduction

Members at AT&T in Summit, NJ and at AT&T Unix Pacific in Tokyo, Japan jointly developed a tty subsystem based on STREAMS, to provide generalized processing of single- and multi-byte characters, such as those used to represent many Asian languages, particularly languages with writing systems related to Chinese. The skeleton tty subsystem, as finally developed, consists of a raw driver, supporting an 8-bit data path, and a line discipline able to handle Supplementary Code Set (SCS) characters of up to 4 bytes in width. This generalized character processing can also handle many European languages with no changes at all, and many Asian languages by adding only specialized STREAMS processing modules for the target language. Section 2 presents an overview of the Supplementary Code Set structure,¹ a necessary prelude to discussions which follow. Section 3 presents line disciplines and control mechanisms, followed in section 4 by a discussion of the Japanese I/O subsystem. Section 5 presents generalizations for SCS processing.

^{*} UNIX is a trademark of AT&T.

¹ SCS may also be seen referred to as *EUC* in various places, notably in Japanese documentation. *EUC* stands for *Extended UNIX Codes*, the name by which SCS is known in Japan.

2. An Overview of the Supplementary Code Sets

Since many languages require more than the ninety-six printing characters of the ASCII character set, various means have evolved for handling larger character sets.² Europe moved to 8-bit characters, and extended the ASCII character set to include such widely-used symbols as é and ù. In much of Asia, where thousands of separate graphic symbols may be necessary for the barest expression of a language, two or more bytes are needed to encode characters. Usually the setting of the high bit distinguishes between codes from different character sets. In some systems (among them the *Shift-JIS* system widely used on personal computers in Japan) there is no *consistent* use of the high bit setting; characters may be encoded by two bytes, only the first of which has the most significant bit set. Such systems have the obvious disadvantage of context sensitivity in distinguishing ASCII characters because they require various kinds of special handling, look-ahead, or backward scanning for such ordinary occurrences as shell meta-characters and ASCII alphabets, making them difficult to work with in the UNIX System.³ Additionally, many codeset systems are defined for or optimized for a particular language, to the exclusion of other languages. This makes for poor portability, and leads to excessive development times or complete re-writes when moving to new languages.

While the world as a whole may in the long run move to a `char` type of more than 8 bits – probably 32 bits – presently such a move is neither economically feasible nor compelling. In the meantime, it was necessary to devise a system that would allow various languages to co-exist on the same machine, with a minimum of software re-write to accommodate them; moreover, the system had to accommodate, to some extent, codesets in current use. The Supplementary Code Sets implemented in JAE 2.0 provide a means of overcoming the portability problems, and create a uniform structure into which various languages can be mapped. The SCS structure itself does not define any codesets other than ASCII, but allows use of various existing codesets. Some of the fundamental design goals behind the SCS structure were:

- to eliminate having to write specialized system code to handle particular languages;
- to develop a system of encoding that separates the actual codesets from their particular graphic representation;
- to remove much of the burden of system-wide codeset requirements from user-level programs; and
- to maximize the re-usability and portability of system code.

The operating system and its associated utilities should be independent of the particular language used by a system's user community. Users should be able to interact with any particular UNIX System implementation in their own language, assuming access to a terminal that can display the appropriate symbols. To some extent the implementation of the SCS structure achieves this goal.

The key to the Supplementary Code Set structure is a distinction between *specific codesets* and the *structural form* of their representation (e.g., as a string of bytes). The SCS structure provides four structural forms (or molds) to handle up to four codesets simultaneously. This seems like an arbitrary number, but it happens to correspond to the maximum known number of codesets needed by a single language – Japanese. The number of available characters can actually be expanded a great deal by merely increasing the number of bytes per

² That these characters are entirely inadequate even for American English can be easily ascertained by observing the frequency with which symbols such as ® • ° ¢ are used in ordinary typesetting. It is amusing to note the stifling effects of such a sparse character set on the development of computing – lack of such a desirable symbol as ≠ for instance has resulted in a plethora of approximations, ranging from != to <>.

³ Shift-JIS may be easily used with JAE 2.0 by the insertion of a codeset conversion module into the I/O Stream. On the input side, a conversion is done from Shift-JIS into the Japanese mapping of SCS, and on the output side, conversion is done from SCS to Shift-JIS. Such a module can be controlled via the `ioctl(2)` calls listed in the next section. It should be noted that since systems like Shift-JIS do not conform to the SCS rules, many programs will break when presented with *files* containing such data – they are not recognizable as legal SCS, which may produce bizarre results, though their use in files is not disallowed.

codeset to three or four. The SCS structure uses two "single-shift" control codes to distinguish characters of the two "least frequently used" codesets of the four. These single-shifts are referred to as SS2 and SS3, represented respectively by hexadecimal 8E and 8F; other 8-bit values represent codes from either the ASCII codeset or from Supplementary Code Set 1. The table below shows the SCS structural form of the four codesets:

Code Set	SCS Structural Form
Set 0 (Primary)	0xxxxxxx
Set 1	1xxxxxxx [1xxxxxxx [...]]
Set 2	SS2 1xxxxxxx [1xxxxxxx [...]]
Set 3	SS3 1xxxxxxx [1xxxxxxx [...]]

Each 8-bit value from Supplementary Code Set 1, 2, or 3 has the most significant bit set, making it instantly recognizable as non-ASCII. This eliminates much of the context sensitivity associated with other encoding systems. Many programs that manipulate character strings only with regard to delimiters, meta-characters, or keywords from the ASCII codeset can operate in a uniform manner on bytes from any Supplementary Code Set. Characters from a Supplementary Code Set may span one or more bytes, with the actual mapping used at any particular time controlled by the value of an environment variable called CSWIDTH.

2.1. CSWIDTH

The CSWIDTH environment variable sets the current attributes of the Supplementary Code Sets in use, specifically, the number of bytes per character stored in memory and the number of columns per character displayed on the output device.⁴ For example, the Japanese environment uses four codesets simultaneously. Codesets 0 and 2 require one column on the output device, but require one and two bytes respectively in memory (including the Single Shift character announcing Set 2).

The CSWIDTH environment variable contains three ordered pairs of digits delimited by colons and commas. Only essential information is encoded – it is assumed that ASCII requires one byte in memory and one column on the output device, so the encoding omits ASCII. The normal setting in the Japanese environment is: CSWIDTH='2:2,1:1,2:2'. The meaning of these pairs are indicated in the following table, and a library routine is provided to read and parse the CSWIDTH variable.

CSWIDTH=X:x, Y:y, Z:z	
Field	Interpretation
X	Set 1, number of bytes total
x	Set 1, screen width
Y	Set 2, number of bytes after SS2
y	Set 2, screen width
Z	Set 3, number of bytes after SS3
z	Set 3, screen width

For any particular language, at some point it is necessary to decide which codesets should be assigned to which of the Supplementary Code Sets. In Japan, this decision was reached through the consensus of a group collectively known as the Japanese UNIX System Advisory

⁴ Here and in the following discussion, *output device* and discussions of display width refer only to video display terminals and letter-quality or dot-matrix type printers. This does *not* include phototypesetters or more sophisticated laser printers with which other, more specialized problems are associated. However, it is interesting to note that the notion of display width is analogous to that of font weights used for proportional spacing.

Committee. They decided to assign JIS C 6226 to Set 1, JIS C 6220 to Set 2, and to leave Set 3 open for "user defined" characters or character sets. JIS C 6226 encodes approximately 7000 of the most frequently used characters. JIS C 6220 encodes a set of one-column (or "half-width") phonetic symbols. Any particular installation can, of course, assign any codeset to any position in the SCS structure depending on their needs, but portability and information interchange between installations would be severely limited if every installation chose codesets arbitrarily.

If the environment lacks a `CSWIDTH` variable (or if `CSWIDTH` has a null value), character processing in both `STREAMS` and system programs is done as if each byte (regardless of the state of the high order bit) were a single character. In other words, absence of `CSWIDTH` is interpreted as `CSWIDTH='1:1,0:0,0:0'`, which covers most European languages *by default*. ISO codesets exist for processing various (unspecified) African languages, Greek, and Cyrillic – all of which can be handled with the default `CSWIDTH` setting. The default setting can also handle codesets for Arabic and Hebrew (each single-byte codesets) provided that the right-to-left movement required for these languages is handled appropriately.⁵

2.2. Multi-byte Processing

Often data must be manipulated as *characters* rather than *bytes*, as in the case of a text editor, which reads characters from the user, and has the capability of backing up and erasing unwanted input. For an editor to be easy to use and intuitive, it should at a minimum allow the backspace key to back up over and remove a single character. Whenever the character is ASCII, this should be one space; in some other code set, the character may span more than one physical column on the output device in a manner similar to the conventional representation of control codes. Backspacing must be handled appropriately in both the storage buffer and on the device.

Backspacing over multi-column characters normally still requires that the output device receive two *cursor-left* (or *backspace*) sequences in order to fully cover the character. When the user depresses the backspace key, the hypothetical text editor needs to determine what character is at the current position and how many columns wide it is on the display. This is easily calculated by looking at the character's first byte, deciding what the character set is, finding the screen-width of that character set in an internal table, and sending the requisite number of backspaces.

As stated in the SCS design goals, user-programs should not be burdened by system-wide external codeset requirements, which are highly variable and intimately tied to physical devices. Not only is such a burden a tremendous waste of time and effort because these requirements can be met once and for all in system code, but it can easily lead to inconsistencies and an explosion of input methods, binding programs to a single language. The SCS structure and `STREAMS` tty subsystem insure that character devices appear to produce SCS codes, as will be shown below.

3. Line Disciplines for Generalized SCS Processing

For user-level programs, the most visible part of the `STREAMS` tty subsystem are the `ioctl(2)` calls defined in the `termio(7)` section of the System V manual. The `termio` facilities, which include setting character width, parity, CR/NL mapping, tab expansion, canonical line input, and so forth are collectively referred to as a *line discipline*. In general, a `STREAMS` module handles most of these functions, but some of them, such as baud-rate switching, may be handled lower in the Stream by the *raw driver*. The raw driver generally performs hardware-dependent functions; however some devices implement larger portions of the `termio` line discipline in hardware, such as CR/NL mapping and tab expansion. The AT&T 3B2 computer

⁵ Processing Arabic and/or Hebrew text seems, on the surface, to not be a difficult problem once an 8-bit environment is assumed; however when a mixture of text from languages with different polarities must be displayed *as it is entered*, proper handling in a *clean* and *intuitive* manner presents an interesting implementation problem.

used for the SCS development work has both types of devices, resulting in two nearly equivalent line discipline modules (called `eld0` and `epld`) intended for use with the two types of devices.

3.1. Processing `termio(7)` Functions

Since `termio` functions are normally split between the raw driver and the line discipline module, these functions are generally handled in the following manner:

- On the way down the Stream, the line discipline module looks at the relevant `ioctl` buffers, setting any modes required for proper operation of the driver below, and passes them on downstream. Both line discipline modules insure that the drivers return all eight bits, but one of the them always additionally insures that the driver (which manipulates an "intelligent" on-board controller) returns single characters as they are typed.
- On the way up the Stream, if the driver acknowledged the `ioctl` call (as indicated by the message type changing to `M_IOCACK`), the line discipline module looks at the relevant `ioctl` buffers, retrieves information which it needs, and passes them on upstream.

This split processing violates one of the normal rules of STREAMS I/O, that the *first* module which recognizes an `ioctl` call must acknowledge it. This rule is impossible to enforce in cases where processing is potentially divided among separate modules. With SCS processing modules in the Stream, it was determined that some modules may have the need to know certain information contained in the `termio` calls, such as the user's current *erase* and *kill* characters. It will be shown below that some of this information is in fact used by the Japanese I/O subsystem.

In a multi-byte environment, some processing required for full implementation of `termio` is relatively meaningless. Despite this, for the sake of compatibility, things such as parity handling and small character widths (such as 5 and 6 bits) were left in. The `stty(1)` definition of `sane` was changed at the same time to include `cs8 -parity`, since this makes more sense in an 8-bit environment.

3.2. Raw and Canonical I/O Processing

In System V, `termio` defines two major states: *raw mode* and *canonical mode*. In raw mode the line discipline module returns each byte as soon as it becomes available from the device. In canonical mode it buffers input until an end-of-line character or EOF is received. During the period of buffering, erase and kill processing are performed on the input line.

When processing multi-byte characters, returning each byte to the user program as the device produces it defeats the purpose because the input device may not produce SCS codes, but may produce some other kind of encoding that must be *translated* into SCS. Moreover, some languages may require that other codes be produced according to the user's specification, which may be a complex process (as seen in the following discussion of the Japanese input system). The production of SCS codes from codes produced by the input device must take place in the input Stream *below* the line discipline module, requiring a class of modules to translate these codes into SCS.

Thus `termio` should affect only the line discipline module and possibly the raw driver but should not affect any input translation modules that may be present in the same Stream. This is essentially a third input state of the tty subsystem as a whole (sometimes referred to as "halfcooked") somewhere between pure raw mode and canonical mode.

With the potential for complex SCS processing occurring in the lower Stream *real* raw mode is almost never used, as it may deprive the user of the ability to provide SCS input and programs may be exposed to raw device codes. To programs that use only `termio`, what looks like raw mode still allows the input of SCS characters produced by whatever modules are below the line discipline module in the Stream. A surprisingly large number of applications can virtually ignore the SCS environment provided they avoid stripping bits, and as long as they don't really care about the *semantic content* of incoming data.

3.3. New `ioctl(2)` Calls for SCS Module Control

A set of new `ioctl` calls handle both multi-byte languages and various types of code conversion modules in single-byte languages. The calls form a small set of primitives for manipulating the state of such modules in general. Conversion and translation modules convert from one type of codeset to another. A good example is the Japanese I/O module: normal Japanese terminals do not process SCS codes directly, but use variants of ISO escape sequences to announce a locking shift into a different codeset. Conversion modules must keep track of the terminal's current output state (ASCII or one of the SCS codesets), and switch it when appropriate; they must also convert outgoing SCS codes into whatever codes the terminal expects (usually this will be some kind of escape sequence).

Some of these calls are needed to manipulate the modules in the Japanese I/O Subsystem described below, and were generalized rather than left as Japanese-specific in order to cover similar functions in Chinese and Korean. It was anticipated that when implementing modules for various languages, a plethora of new calls might be created to control the modules' states; these primitives are an attempt to replace at least some of the anticipated language-specific controls.

Command	Effect
EUC_WSET EUC_WGET	Send SCS widths to line discipline Get SCS widths
EUC_MSAVE EUC_MREST	Save state & return to ASCII Restore state if saved
EUC_IXLON EUC_IXLOFF	Turn on input translation Turn off input translation
EUC_OXLON EUC_OXLOFF	Turn on output translation Turn off output translation

Line discipline modules interpret the `EUC_WSET/EUC_WGET` pair of calls. The codeset widths are used mainly for erase processing in the line discipline module. After they are acknowledged, control messages are routed downstream to inform other modules in the Stream of the codeset widths in use. A program called `eucset` takes a `CSWIDTH` specification as argument (or obtains it from the environment if no arguments are given) and sends the `EUC_WSET` command. It was determined that the shell should not be modified to automatically inform the line discipline module of `CSWIDTH` changes, since users may desire to use a different specification in some processing than in the terminal input Stream,⁶ for example, to do background processing in one language, while otherwise interacting with the terminal in another.

One simple rule governs the use of the last six calls listed in the table above: all *modules* that process SCS and use input or output conversion must act appropriately on receipt of the commands, and all *drivers* that receive the calls must acknowledge them. Thus, the calls are propagated through the Stream, acted upon by modules which may be affected, and passed on to neighboring modules. This also allows user-level programs to determine whether or not they are running on a system that supports SCS – on non-SCS systems, the calls will all be negatively acknowledged, and the `ioctl` call will return -1, with `errno` set to `EINVAL`.

The `EUC_MSAVE/EUC_MREST` pair provides a means of temporarily switching the state of input translation modules, such as the module described below for Japanese input. When an `EUC_MSAVE` call is received by a module with "modes" (refer to the next section) it is expected to save its current mode and state, and to return to ASCII mode; upon receipt of an `EUC_MREST` command, it is expected to restore the saved state. This pair is used by `vi(1)` in

⁶ Normally this is not a burden, because most users set up their environment once at login time, and don't change settings.

switching between *command* and *input* modes.⁷ Upon receipt of an escape character to enter command mode, *vi* sends an `EUC_MSAVE` command so that the mode is switched back to ASCII, since *vi* requires that all commands are ASCII. (No switch is performed by modules if their mode is already ASCII.) When switching back to input mode, *vi* restores the previous mode so users can continue with insertion in the same mode as before.

The final two pairs of calls are for switching input and output conversion. Normally, these are not needed by application programs, but they may be used, for example, to download font patterns into a terminal. Japanese input devices normally provide a means of downloading user-definable characters for subsequent display. Since the downloaded information may be binary, output conversion of the data could produce unintended results; thus, a font downloading program should inhibit output conversion during the download process.

In addition to the calls described above, all the drivers and modules of JAE 2.0 obey the convention that `M_CTL` messages within the Stream take exactly the same form as `M_IOCTL` messages. That is, they all consist of one message block pointing to a data block containing an `iocblk` structure, with associated data as for `M_IOCTL` messages. The message type is `M_CTL`, and the `ioc_cmd` field of the associated `iocblk` is set to the desired command.⁸

4. The Japanese I/O Subsystem

Three separate symbolic systems are used together to write the Japanese language. Two of them are phonetic systems called *Hiragana* and *Katakana*; the third (*Kanji*) is a system of characters imported from China.⁹ The three systems are used in a completely integrated fashion in the language, and each has its special place and special usage in a manner somewhat analogous to CASE and *font* distinctions in English.

Japanese input processing differs from English input processing in one fundamental way: the characters used in the language cannot necessarily be *directly* entered by a user from any keyboard with a reasonable number of keys. Characters may be *described* by a number of key-strokes varying from one to three or four; their actual codes are manufactured programmatically. This difference actually applies to a number of different languages, whose speakers have developed various means of coping with the difficulty, some of which will be described below.

4.1. Overview

The character sets of languages with writing systems related to Chinese are generally large (on the order of several thousand characters). They are based on a number of basic elements (on the order of a few hundreds) called *radicals*, comprised of a number of *strokes*. Characters may be described in terms of these radicals and strokes, and by the internal relationships between the various radicals or strokes within the character. A character is, in general, more analogous to an English word than to a letter, in that the character carries not only a sound, but a specific meaning as well.¹⁰

In the People's Republic and Taiwan, the trend has been toward entering characters by code number (where they fall in the codeset) and more recently toward entering characters by describing how they *look* by entering a small number of *strokes*. With many Chinese input systems, the basic character is chosen by typing a four-digit number, which is turned into a character code and sent to the host machine. The Chinese systems based on strokes or radicals are very similar in principle to the Japanese system described below. Generally, the work is done in the terminal, which may itself be a general-purpose computer. Alternatively it could be done by

⁷ The version of *vi* discussed here is that of JAE 2.0, which has modifications for handling multi-byte and multi-column characters.

⁸ These terms are discussed in the *Streams Programmer's Manual*.

⁹ Some of the *Kanji* differ from their counterparts in the People's Republic and Taiwan, since they have evolved separately in the 1000 years since their introduction.

¹⁰ In the Korean *Hangul* system this is not strictly true, but the constructional principle is the same.

a relatively simple input translation module in the I/O Stream. In both of these systems, the user bears the most of the burden of describing the character to the machine.

The Japanese have generally gone in the opposite direction, wishing to have the machine bear most of the burden: the user types phonetic symbols, and the machine presents a number of choices (called *candidates*) that match the pronunciation. In the most sophisticated systems, the machine actually tries to parse the phonetic input, choosing characters that seem to fit best. These systems can reach high levels of accuracy, requiring the user to make very few corrections or choices.

Japanese input sometimes takes on tremendous complexity. Personal computers (which may be used as terminals) generally have built-in translation mechanisms which can be quite sophisticated; for these, no special processing is required inside the UNIX System. Terminals, however, are quite unsophisticated, and to be usable at all, require a great deal of support from the host. Most modern Japanese terminals use a minor variant of the QWERTY keyboard; they may have an additional locking shift key that turns on the high bit of the bytes sent to the host. This key is called the *kana lock* key. The keycaps are inscribed with the *Katakana* phonetic symbols, in addition to their English legends. By depressing the kana lock key, these phonetics may be entered as single bytes with the high bit set, to distinguish them from ASCII.¹¹ The devices accept from the host either ASCII, JIS C 6220 *kana* (which are single-width phonetics), or JIS C 6226 *Kanji*. *Kanji* requires a separate output mode, entered via a locking-shift type escape sequence; the *Kanji* codeset actually includes all of the *Hiragana* and *Katakana* characters as well as special symbols, and the Chinese (*Kanji*) characters themselves. The *Kanji* and phonetics are usually twice the width of ASCII on the display.

4.2. Input Modes

Entry of Japanese characters requires a STREAMS module to translate input. At any given time, the module is in one of several *input modes*, in which it manipulates input key-strokes in various ways. Single-width *kana* may be entered directly, by depressing the kana lock key. In the proper mode, these may be translated into *Hiragana* or *Katakana* by the module. Additionally, the user may enter Romanized equivalents for the pronunciation of the phonetic symbols, and let the module translate them into the actual symbols. Altogether, the module has five input modes for generating various types of phonetic, multi-byte alphabetic, and special symbols. In some cases, phonetic symbols are sufficient, and the user sets the module into a mode where each phonetic symbol is sent upstream as it is generated. In many cases, however, phonetic symbols are not sufficient, but must be translated into *Kanji* characters. In these cases dictionary access is necessary to perform the translation.

The system keeps one or more dictionaries which key phonetic strings to strings of *Kanji* with a given pronunciation. There may be many *Kanji* with the same pronunciation, so the user must make a choice among the various candidate strings. This requires some method of presenting the candidates to the user, a method of storing both the input phonetic string and the candidates, a method of determining the user's choice.¹² All of these functions are handled by the Japanese I/O module in a *dialog* with the user. STREAMS modules within the kernel handle the entire dialog; only the result of the user's choice is passed upstream.

¹¹ This discussion is a highly simplified view of the actual situation, but, unfortunately, space is not available for a detailed discussion covering all of the possibilities. The focus here is on how these devices are actually used in the present system.

¹² This method is very basic. Methods of much greater sophistication are possible, and even likely in the future, using STREAMS modules. The present method does not pretend to be *the best* method for Japanese, merely *a method* that provides slightly more than minimum capability. It has the advantage of being more widely applicable and highly generalized, as will be shown below.

4.3. The Candidate Dialog

A *dictionary daemon* looks up characters in a dictionary. On a given machine, there may be one or more daemons running. Each daemon is initially executed with zero or more associated system dictionaries, and can open a private dictionary for each user. At startup time, when the translation module is pushed into the input Stream, an initialization program searches the environment for a variable called `DICTD`, containing the name of the user's private dictionary. This is sent via `ioctl` to the translation module, which in turn informs the dictionary daemon. The actual inter-module connections to accomplish this are discussed below, after an outline of the candidate dialog is presented.

The dialog between the user and the translation module for candidate choice proceeds something like this: the user enters a command to begin saving the phonetic text to be looked up. While saving text, the user may use the normal erase and kill characters (obtained through `termio` calls) for editing the input. At the end, another key tells the module to send the data to the dictionary daemon. The translation module builds a packet with information identifying the originator, and ships it to the daemon. A single daemon may service several users, which helps reduce the number of processes that would otherwise be required, and reduces the number of open files since each daemon will have one or more system dictionaries open at all times.

While awaiting a reply from the daemon (the process being somewhat asynchronous) the module disables its own queue, allowing it to back up.¹³ Users will normally not enter many keystrokes while waiting for the candidates to be displayed. When the daemon's reply is received, the first candidate is displayed at the current cursor position, in some form of highlighted mode. This process requires that the module know how to save and restore the cursor (a capability of all supported Japanese terminals). The user may press various keys to display the next candidate in-place, skip backward, abandon the selection, or enter *guide-line* mode, in which candidates are displayed, several at a time, on the bottom line of the screen. All supported terminals can either remove lines from the scrollable region of the screen, or can display candidates on a status line. When the user chooses a candidate, the cursor is returned to its original position, and the candidate is sent upstream.

4.4. Module Connections

The actual "plumbing" involved in the translation module to dictionary connection is not exceedingly complex. A multiplexor could have been used, but it has several disadvantages: (1) it requires another module, with its associated overhead, (2) it requires another type of inter-module communication between the translator and the multiplexor, (3) multiplexing configurations are more difficult to build and dismantle, and are generally not suited to the dynamic environment desired. A multiplexor is also far more generalized than the two-way conversational connection necessary for dictionary access.

A STREAMS device, called `/dev/dict` acts as an intelligent switch between translation modules and dictionary daemons. It is a language independent switch, operating on a simple packet protocol between modules. The dictionary daemon opens this device, and announces itself as a daemon, allowing the dictionary driver to dispatch requests to it from translation modules. Translation modules may connect to it through a "side door" (see Figure 1, at the end of this paper). The upper queues of the driver connect directly to the Stream Head. The driver passes requests, consisting of control packets with attached data, between the translation module and the dictionary daemon. The data packets may contain any SCS data, as null-terminated strings. The daemon processes requests from various users as the requests are read at the Stream Head (through `getmsg` and `putmsg` system calls). The daemon must keep track of each user's state, especially in the event of an extended dialog - there may be a large number of candidates, and the daemon sends only a limited number in return for each request;

¹³ Error recovery mechanisms are also in place, allowing recovery from situations where the daemon has been killed or is otherwise inaccessible.

sometimes the batch of candidates is complete, at other times several batches may be necessary, as the number of candidates for a particular phonetic string varies between one and about two hundred. The daemon attempts to order these by frequency of occurrence, and can learn during a session, putting more frequently chosen candidates (per user) at the beginning of the list. This helps reduce extended dialogs over the course of a session, as many words are frequently re-used.

Aside from the obvious open/close and request/result pairs, the packet protocol has a set of primitives for retrieval of candidates using two different matching algorithms (only one of which is actually implemented) and selecting different private dictionaries at will. The protocol, as mentioned, uses SCS strings. The daemon's dictionary files have headers indicating their encoding CSWIDTH, and all files opened by a particular daemon must match. The present daemon actually implements only Japanese dictionaries, but the protocol can accommodate any potential SCS mapping. The daemon is divided into two parts, the front end and the back end. In this particular release, the back end, which performs the actual indexing operations, is optimized for Japanese, while the front end remains a full SCS processor.

The initial connection between the dictionary driver and the translation module depends on a "well-known" routine, defined in a common header file. Once the connection is made, the driver and module call each other directly whenever they have data for one another, since relevant addresses are exchanged in the process. The dictionary driver "peeks" at some types of packets, particularly those that make initial connections. Since there may be several daemons, the initial connection request must identify a daemon. Daemons are called by name. A daemon takes its name from the command line, and informs the dictionary driver of it during the initialization sequence. To cater to groups of users with differing needs, the same language may have several daemons, potentially with different dictionaries and different names.

4.5. Further Explorations

This type of dictionary interaction can work with many other languages, even those without a strict need for it. The Korean language, for example, uses both *Hangul*, which can be manufactured by a translation module, and Chinese characters, which cannot. A prototype *Hangul* translation module has been developed which provides access to these Chinese characters, in common use and defined in the character set, but which cannot otherwise be directly generated by the module (other than by code number).¹⁴ If Chinese terminals without on-board translation capability come into widespread use, the same type of module could be used for Chinese to provide phonetic, code number, and stroke-based input.

One of the current explorations underway is to separate the translating portion of these modules from the dictionary access portion. While the resulting modules must be tightly coupled, and conform to a common protocol, the dictionary interaction is localized in a single module, rather than being replicated in each translation module, which will significantly reduce code space when multiple languages are used on the same machine. At the same time, the daemon's back end can easily be uncoupled and generalized for SCS, so that all languages can be serviced by the same daemon – this reduces the number of active text segments when running multiple languages.¹⁵

There are other uses for dictionaries, even for languages that have no real need for them. They can be used to provide a kind of "aliasing" facility – really, this is what has been implemented for Japanese – or "macro expansion", or may be used rather like a large number of programmable function keys, with reasonable invocations. They could also be used in learning another language, if coupled with a bi-lingual dictionary to return *word definitions*; they could also have some applications in on-line translation of documents. In all of these cases, only the

¹⁴ There is, however, no actual *dictionary* for use with this module.

¹⁵ Obviously, however, it is useful to continue having separate invocations of the daemon for different languages, which would presumably employ separate system dictionaries, and possibly different SCS mappings.

translation modules must change; neither the dictionary daemon nor the dictionary driver need change.

5. Working with Generalized SCS

Once an implementor grasps the concept of the multi-byte character, it is not exceedingly difficult to extend character processing to include generalized SCS processing. Given the right tools, such as a library of SCS character routines, it becomes even easier. The JAE 2.0 release includes such a library.

While changes were made to allow System V to work more smoothly in an international environment, these changes are relatively invisible to users working only in English: the command names remain the same, and with ASCII, they work the same way. The changes came in handling full 8-bit and multi-byte characters. Filename expansion and programs that deal with regular expressions all now deal with multi-byte characters.

5.1. Basics of Working in an International Environment

When working in an international environment, the foremost rule is to process all eight bits of a character. Bits should never be stripped, and bytes (the `char` data type in C) should usually be viewed as 8-bit unsigned quantities. When there is any doubt about the possibility of sign-extension, characters should be declared as `unsigned char`. Programmers should exercise caution in using bytes as indices into arrays (128 entries are not enough).

There are several new facilities in the international environment, such as the ANSI C Localization Library. This is a library based on ANSI X3J11/86-151, a proposal for standardization of the C Language. These routines include capabilities to define, for single-byte languages, character classes and numeric representations. Also included are date and time conversions and 8-bit collating sequences. By calling the `setlocale` function, these can be set up for automatic use by the standard C libraries. New routines were written to convert date and time into locally acceptable strings for various languages. These routines are used by programs such as `date` and `ls`. Where possible, using the `CSWIDTH` environment variable, if defined, can help insure that character processing is compatible with the user's language. It also helps in application programming to remove embedded strings from programs, putting them in a separate file if possible. This allows them to be moved to new languages more easily.

5.2. Methods of Working with Multi-byte Characters

Programmers can work with multi-byte characters in many ways. Some of these will be outlined here. In general, the problem for programmers is to recognize multi-byte characters as single entities, and process them appropriately. This includes handling strings correctly by keeping bytes of a character together, rather than allowing them to be broken apart. For example, it is fatal for a text editor to break a multi-byte character across lines. Fouling character boundaries can lead to errors in output conversion state-transitions, misplaced escape sequences, and most alarmingly to the user, garbage on the screen and in files. Some of the available techniques are briefly discussed below.

ASCII strings. For application programs that only care about dividing arguments and possibly recognizing ASCII delimiters, such as spaces and tabs, use of normal string processing is perfectly acceptable. Multi-byte characters can be directly printed, copied, concatenated, and so forth using the normal output routines.

Arrays of Indices. One common string-processing technique is to use two arrays. One array holds the actual string of bytes. A second array of either character pointers or of short integers can be used to point to the first bytes of the characters in the string. Indexing through the index array gets a pointer or index to the first byte of the current character. This technique is particularly useful with backspace processing since it is easy to move back one character, obtain its width both in the buffer and on the screen, and remove it.

Markers. A separate array of markers may be kept, with character attributes for each byte of a string. Keeping a bit associated with all bytes that are *first bytes* of a multi-byte character allows one to move quickly around the array to the next or previous character. This might be used with a text editor to keep the cursor positioned properly.¹⁶

Process Codes. In JAE 2.0 there is a library, called the Wide Char Library, that allows manipulation of multi-byte characters as single constant-width units, referred to as *process codes*, since they are used strictly within user-level processes, never in I/O or in files. Generally these entities are cast into an `unsigned short` or `unsigned int` type. The type `wchar_t` is a `typedef` for one of these. Routines exactly paralleling the standard C string and character functions (such as `strcmp` and `getc`) with some new additions are provided for manipulating these process codes. Sixteen bit process codes for various SCS mappings are shown in the following table.

Code Set	SCS Code	Process Code
0	0xxxxxxx	000000000xxxxxxx
1	1xxxxxxx 1xxxxxxx 1xxxxxxx	100000001xxxxxxx 1xxxxxxx1xxxxxxx
2	SS2 1xxxxxxx SS2 1xxxxxxx 1xxxxxxx	000000001xxxxxxx 0xxxxxxx1xxxxxxx
3	SS3 1xxxxxxx SS3 1xxxxxxx 1xxxxxxx	100000000xxxxxxx 1xxxxxxx0xxxxxxx

Linked Lists. When extensive processing, such as insertion and deletion, must be done on strings of multi-byte characters, singly- or doubly-linked lists of character cells or process codes may be used. This is an easy method for insertion and deletion, but it may take more processing to obtain a null-terminated string, and to do output.

5.3. A Basic SCS Input Algorithm

The following algorithm compactly illustrates processing that might be used to retrieve characters as multi-byte entities from the standard input stream (this is *not* actual code used in JAE 2.0). A two-dimensional array, `wtab`, contains the screen widths and *actual* memory widths for each of the four Supplementary Code Sets. It returns the number of bytes read from standard input (or zero for EOF), and sets the variable pointed to by `codeset` to the codeset of the retrieved character. Using one of the techniques described above, an array of these characters is relatively easy to manage. When using an array of index pointers one could set the index then call this routine passing the address in the character string at which to put the bytes. For printing characters, the return value indicates the number of bytes to move forward. Of course, if backspace processing is needed, one should examine the returned character before inserting it into the array.

¹⁶ In Asian languages with multi-column characters, the general expectation is that the cursor will always be positioned over the left-most column of a multi-column character.


```

#define HIBIT      0x0080
#define CH_MASK    0x00FF
#define SCREEN     0
#define WIDTH      1
/*
 * Retrieve SCS character from standard input.
 */
getSCS(s, codeset, wtab)
    unsigned char *s;          /* where to put SCS char */
    int *codeset;              /* codeset of SCS char */
    unsigned char wtab[];      /* Code Set width table: 2 x 4 array */
{
    register int c;             /* for 'getchar' */
    register int bytesleft;     /* bytes yet to come */

    if ((c = getchar()) & HIBIT) { /* is SCS? */
        switch (c) {
            case EOF:           return 0;
            case SS2:           *codeset = 2; break;
            case SS3:           *codeset = 3; break;
            default:            *codeset = 1;
        }
        *s++ = c & CH_MASK;
        bytesleft = wtab[WIDTH][*codeset];
        while (bytesleft-- > 0) {
            if ((c = getchar()) == EOF)
                return 0;
            *s++ = c;
        }
    }
    else { /* no, it's ASCII */
        *codeset = 0; *s = c;
    }
    return(wtab[WIDTH][*codeset]);
}

```

6. Development Tools

Obviously, the tty subsystem was developed with tools. Several tools initially used in development made their way into the real world. Two trivial programs, `ilook` and `ifind`, are very useful for looking into the Stream from user level. The program `ilook` returns the name of the top module in the Stream, and `ifind` searches the Stream for a named module, returning its name if found. A program `setterm`, for generalized Stream manipulation, became indispensable in the early stages of development. The Tokyo-based development team often had ten or so different modules compiled into the kernel, some of which didn't always work correctly. Keeping separate programs to pop the current set of modules and to push a new set quickly became unwieldy. The `setterm` tool uses a configuration file whose labelled entries denote sequences of function calls. These sequences allow one to "push" and "pop" modules conditionally or unconditionally, look at the top module in the Stream, run initialization programs, save and restore line discipline states.

During the very early stages, drastically errant modules posed a constant problem. This situation led to the development of a crude sort of dump module (which didn't make it into the real world!). This dump module printed information on every message passing along its queues, allowing developers to actually see on the console what messages and data were being passed between modules. Unfortunately, the system console was *completely* unusable while the dump module was in use, which didn't prove very useful in debugging the console driver. Its major advantage was that it could be popped, dynamically eliminating tracing when it wasn't needed.

Some other modules developed included an interpreter for function keys. This allowed users to map function keys to otherwise unused characters, and thus avoid collisions between those special characters for translation modules and those used by text editors, such as EMACS, and with Korn Shell line-editing modes. A Latin Alphabet module supports entry of European codesets with an ASCII keyboard on an AT&T windowing terminal. This same terminal also implements Japanese, and windows may be manipulated individually with respect to fonts and STREAMS modules. This allows European and Asian languages to be used simultaneously on a single terminal (see Figure 1 at the end of this paper).

7. Restrictions and Open Issues

Several facilities of UNIX Systems should remain untouched, no matter how far one takes internationalization. Of these, login names and passwords stand out. Using multi-byte characters for these clearly reduces inter-machine connectivity, especially in multi-lingual

environments, when different machines each speak many languages, but share no common language. As the *lingua franca* for UNIX Systems ASCII should remain the basis for international connectivity.

Two areas have yet to be solved, and present some very sticky problems. These are (1) system messages, and (2) collating sequences and character classes. System messages are still in English. The ANSI C Localization library solved the collating sequence problem for 8-bit languages, but leaves the problem for multi-byte languages in question. Unfortunately, the solution for single-byte languages fails to provide any guidance for the many problems of multi-byte collating sequences and character classifications.

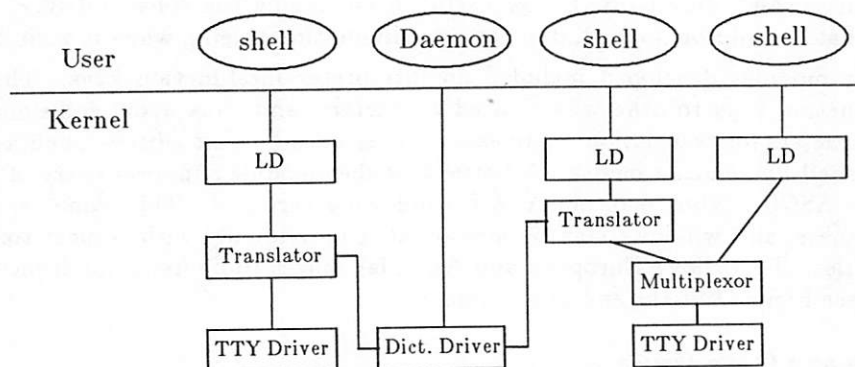
Acknowledgements

Special thanks are due to many individuals: Dick Hamilton and Tsuneo Inada, indefatigable members of the Tokyo-based team; Tom Butler, Hari Pulijal, Curt Schimmel, and Bob Zarow of AT&T Summit Facility, developers of the original STREAMS drivers and modules for the 3B2; Dave Korn and Warren Montgomery of AT&T Bell Laboratories, developers of, respectively, KSH-I and EMACS for multi-byte processing; the members of the teams at Summit who produced Release 3.1, upon which all else rests; the team at AT&T Unix Pacific, who produced multi-byte versions of major tools; Garrett Long, who helped with drafts of this paper; and Larry Crume, whose leadership and vision made this project possible.

References

- [1] International Korn Shell (KSH-I). AT&T Unix Pacific Co., Ltd., 1986.
- [2] Japanese Application Environment Release 2.0 Product Overview. AT&T Unix Pacific Co., Ltd., 1987.
- [3] Korn, D. G. Introduction to KSH (Issue 3).
- [4] Montgomery, Warren. An Interactive Screen Editor for the UNIX System (Issue 2).
- [5] Montgomery, Warren. Adapting a Screen Editor for International Character Sets.
- [6] Pike, R. The Blit: A Multiplexed Graphics Terminal. AT&T Bell Laboratories Technical Journal, Oct. 1984 (Vol 63 No.8 Part 2).
- [7] Ritchie, D. M. A Stream Input-Output System. AT&T Bell Laboratories Technical Journal, Oct. 1984 (Vol 63 No.8 Part 2).
- [8] System V Interface Definition (Issue 2). AT&T, 1986.
- [9] UNIX System V Release 3; STREAMS Primer. AT&T, 1986.
- [10] UNIX System V Release 3; STREAMS Programmer's Guide. AT&T, 1986.

Figure 1. Module Connections. On the left is a terminal configuration with translation module and dictionary connection; on the right, a multi-window terminal configuration with two windows processing different languages.



Automatic Error Recovery in a Fast Parser

Robert W. Gray
Computer Science Dept 430
University of Colorado
Boulder, CO. 80309-0430
bob@boulder.COLORADO.EDU

ABSTRACT

Although parser generators have provided significant power for language recognition tasks, many of them are deficient in error recovery. Of the ones that do provide error recovery, many of these produce unacceptably slow parsers. I have designed and implemented a parser generator that produces fast, error recovering parsers. For any input, the error recovery technique guarantees that a syntactically correct parse tree will be delivered after parsing has completed. This improves robustness because the remaining compilation phases, such as semantic analysis, will not have to deal with infinitely many special cases of incorrect parse trees. The high speed of the parser is a result of making the code directly executable and paying careful attention to implementation details. Measurements show that the generated parser runs faster than any other parser examined, including hand-written recursive descent parsers. The cost of this fast parser with error recovery is a slight increase in space. Although this particular generator requires LL grammars, the ideas can be applied to generators taking LALR grammars. Furthermore, we give the transformations that allow one to transform many LALR grammars into equivalent LL grammars.

1. INTRODUCTION

Although parser generators produce code to recognize the structure of input, many such as YACC [Johnson1979] do not provide syntactic error recovery. The PGS [Dencker1985] tool generates an error recovering parser, but one that runs so slowly that it is a major bottleneck in the overall analysis [Gray1985]. This efficiency penalty is severe enough to dismiss PGS as a useful production quality tool, and compiler writers are forced to do without automatic error recovery. In this paper, I describe the design and implementation of DEER, a tool that produces fast parsers with error recovery. The high speed is achieved by producing directly executable code as opposed to the common practice of producing tables that are interpreted at runtime. The idea of direct execution is not new; recursive-descent parsers have always had significant speed advantages over table driven parsers. The significant contribution of DEER is that it generates from a grammar, a fast parser, with error recovery. Complete details of the ideas presented herein and the code which implements them are contained in *Generating Fast, Error Recovering Parsers* [Gray1987].

The most important part of a compiler in terms of user interface is error recovery. It is unacceptable to merely announce the first syntactic error of an input program and quit. All syntactic errors should be reported when a compiler is run; furthermore, semantic errors should also be reported even if syntactic errors are present. Clearly, a simple syntactic error such as missing punctuation should not preclude semantic error processing. The difficulty here is that

semantic analysis must have a syntactically valid parse tree¹ in order to reliably proceed. Ad hoc error recovery cannot guarantee that a correct parse tree will be passed to semantic analysis — there are far too many special cases that need to be handled. This is why we consider a YACC generated parser to be deficient: when it detects an error a user written recovery routine is called. Often, users of YACC will not take the time to write an error recovery routine and in this case the parser will terminate at the first error. Implementing complete and reliable recovery is involved and tricky; therefore, it should be automated. We desire a formal recovery technique, one based on sound theory. The same technique that Röhrich [1980] has implemented for PGS is used for DEER. The recovery is good for most syntactic errors although there are cases where hand tuned recovery will do better. Still, because of the guarantee of correctness, plus the *no work required of the programmer* feature, automatic error recovery is the method of choice.

Section 2 describes how very fast parsing can be achieved through direct execution. Section 3 presents the recovery technique and how it has been incorporated into the very fast parser. Performance measurements in Section 4 confirm the advantage of direct execution. Section 5 discusses grammar transformation.

2. DIRECTLY EXECUTABLE PARSER

This section addresses the problem of generating a very fast parser. We know that recursive descent parsers are fast. These parsers are based on LL(1) grammars, so it should be possible to *generate* a parser that runs as fast as a recursive descent parser. As it turns out, a DEER generated parser runs significantly faster.

The key insight for high speed parsing is that the parser should be directly executable. Recursive descent parsers are directly executable — there is a procedure to recognize each production of the grammar. Transitions of the automaton are often implemented with jumps, for example a PASCAL parser might contain

```
if ( lookahead == 'BEGIN' )
    begin_processing_code
else if ( lookahead == 'IF' )
    if_processing_code
...
```

On the other hand, most parser generators (including YACC, SYNPOT, PGS, Bison) produce tables that must be interpreted. The automaton of such a parser enters the next state by looking in a table, and performing the action.

table : state \times symbol \rightarrow action

On the average, this takes at least 3-4 times longer than direct execution. Therefore, to achieve very fast parsing, DEER produces executable code, not tables.

Figure 1 gives a fragment of a DEER generated parser. It is similar to an assembly language listing of a recursive descent parser. The code for each case corresponds to code of a procedure in a recursive descent parser — one per non-terminal of the grammar. A call to these pseudo procedures, is implemented as the sequence '*PU(); goto L;*' and a return is *goto pppop*. Notice that the case labels are compact. This is one of the implementation details that can yield an improvement in execution speed of up to 25%. *IF_NOT* is a carefully coded macro that generates a single machine instruction to test whether the lookahead symbol *la* is in the set referenced by the first argument. For example, D2 might be the set of terminal symbols that

¹ Often, there is no reason to form an explicit parse tree, instead a prefix linearization may be more appropriate (see *connection points* in [Waite1983]).

initiate statements: { *IF, BEGIN, WHILE, REPEAT, ...* }. When the parser cannot accept the lookahead in its current state, a call is made to `parseErr`, which carries out the error recovery. When it returns, normal parsing resumes.

```

goto L0;
pppop:      switch (--*DEPSp) {
case 0:      break;
              L0:  IF_NOT(D2,la) la=parseErr(la,L0);
              L1:  PU(1); goto L14;
case 1:      L2:  goto pppop;
              L3:  if (la != int) goto L6;
              L4:  la = nextterm( );
              L5:  goto L13;
...          ...
case 3:      L25:  if (la != ]) la=parseErr(la,L25);
              L26:  if (la != of) la=parseErr(la,L26);
              L27:  IF_NOT(D2,la) la=parseErr(la,L27);
              PU(4); goto L14;
case 4:      L28:  goto pppop;
              }

```

Figure 1. Directly Executable Parser

At first glance, the directly executable parser appears to have a structure similar to an interpretive parser — a big switch statement. However, an interpretive parser requires a loop iteration for every state change: the directly executable parser makes most transitions with sequential execution and goto statements. A switch statement is required for only a fraction of the cases.

Figure 2 shows the process of building both a YACC and a DEER parser. Again, the major difference is that *y.tab.c* is mostly a file of tables that will be interpreted whereas *files* is mostly directly executable C code.

3. ERROR RECOVERY

Now that we have obtained high parsing speed, the problem is to incorporate error recovery without diminishing parsing speed. The error recovery should be automatically generated from the parsing grammar without any extra effort from the user of DEER. The quality of its syntactic error messages is an indication of the user friendliness of a compiler. In the best case, the user is immediately led to all syntactic errors of his program. In the worst case, if error recovery operates incorrectly, the compiler may crash, leaving the user helpless.

Section 3.1 gives a motivation for automatic error recovery and brief overview of existing techniques. Section 3.2 informally presents the ideas behind the DEER error recovery method.

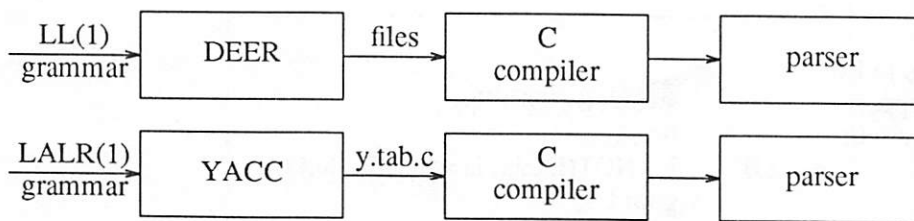


Figure 2. Steps to build YACC and DEER parsers

3.1. Overview

As a parser operates, it consumes input, token by token. The consumed input, also called *accepted* input, drives the parser into a particular state. Deterministic (non-backtracking) parsers never accept a token that cannot legally continue what has already been accepted. This is one of the principle merits of LL and LALR parsing techniques — they are guaranteed to detect errant tokens as soon as they are encountered.

The user should receive as much information as possible from each compilation attempt. It is unacceptable just to detect the first error and quit. The parser should repair errors and continue parsing. Finally, it should deliver a valid parse tree or connection sequence to the rest of the compiler. If the parser tries to recover but delivers a faulty parse tree, the remaining phases of the compiler could crash and leave the user helpless.

Gries [1976] gives an excellent annotated bibliography for error handling. Also, Horning [1976] presents an overview of various techniques of error handling. There are many strategies a parser can employ for error recovery: panic mode, phrase level, error productions, global, and automatic.

One of the simplest language independent recovery techniques is the *panic mode*. When an error is detected the input is skipped until one of a predefined set of "special" symbols such as *begin* or ";" is encountered. The parsing stack is popped until the special symbol can be accepted. Unfortunately, this method has many shortcomings. It frequently results in deleting large portions of the source text. In addition, semantic information depending on the erased part of the stack becomes inconsistent. Finally, the set of special symbols must be determined by hand.

On detection of an error, *phrase-level* recovery makes a backward move in the parse stack and a forward move in the remaining input. This isolates a phrase which is likely to contain the error. Then a weighted minimum distance correction is carried out at the phrase level.

Joy, Graham and Haley [Graham1982a] use *error productions* for their production Pascal compiler. First the compiler writer needs to predict the most likely kinds of errors expected. Then error productions must be written by hand. It is impossible to foresee all error conditions. In practice, the parser must be exercised to see how well the error recovery works. Further tuning is likely to be needed. There is another shortcoming of this method: the added error productions could make the grammar ambiguous.

Global error recovery attempts to find the smallest set of changes that will make a given program syntactically correct. It is impractical from the perspective of efficiency due to the exponential number of corrections that must be considered. The smallest set of changes if there

are more than two errors is to enclose the error portion with comment brackets. This is usually not a desirable recovery.

Röhrich [1980] has implemented *automatic* construction of error handling parsers for LALR(1) grammars, and Fischer [1980] has done it for LL(1) grammars. The technique is based on a sound theoretic foundation. The resulting parsers are capable of correcting all syntax error by insertion and/or deletion of tokens to the right of the error location. Therefore, no backtracking is needed, and the output of the parser always corresponds to a syntactically valid program. This contributes significantly to the reliability and robustness of a compiler. The speed of parsing correct parts of a program is not affected by the presence of the error handling capability.

Röhrich's technique of automatic error recovery was chosen for incorporation in the directly executable parser because it automatically derives the error recovery directly from the grammar. Many other techniques require the manual specification of error recovery.

3.2. Automatic Error Recovery

In this section, I give a high level, intuitive presentation of automatic error recovery. To affect error recovery, the parser will delete zero or more tokens and generate zero or more tokens, then normal parsing will resume. The simple grammar of Figure 3 will be used for a few examples.

type	→	simple
		[^] id
		array [simple] of type
simple	→	int
		char
		num .. num

Figure 3. Sample grammar

In the following input, the errant token which we will call t is the second left bracket.

array [[int] of char

Our recovery technique will delete the second left bracket and resume normal parsing. For input,

array [int of char

the errant token "of" cannot be accepted in the current parse state. However, after a close bracket is inserted, then it can be accepted. For the input

array [[char

the second left bracket will be deleted, and the generated string of tokens "int] of" will be inserted.

A complete and formal description of the error recovery technique can be found in [Gray1987, Waite1983, Röhrich1980]. For the motivated reader, the rest of this section provides the essential details.

A parser for language L will *accept* input strings (i.e. programs) in L . Let T be the set of terminal symbols of the language L ; then $T^* - L$ is the set of all erroneous programs. Let $\omega t \chi$ be an erroneous program, where ω is an initial string that is syntactically correct and has been accepted by the parser and symbol t cannot be accepted by the parser. The rest of the program is the string χ . We say that t is a *parser-defined error*.

If $\omega t \chi \in (T^* - L)$ is an erroneous program with parser-defined error t , then to effect recovery the parser must alter either ω or $t \chi$ such that $\omega' t \chi \in L$ or $\omega t' \chi' \in L$. Alteration of ω is undesirable since it may involve undoing the effects of previous actions. It is too expensive to retain information in case backtracking is needed. Thus, we consider the alteration of only t and χ .

In more detail, the error recovery works as follows: A fixed terminal symbol $f(q)$ is associated with each state q of the parser. When an error is detected, the parse stack is copied. Then a "continuation parse" is carried out using the copied stack and $f(q_i)$ as input at each state q_i . In addition the set of allowable terminals (*Director set*) of each state q_i is added to the *anchor* set which is the set of all terminals that could be accepted during this continuation parse. The function f is chosen such that this process would terminate the parse rapidly, driving the parser through states q_1, \dots, q_n . Next zero or more of the actual input symbols are discarded until an input symbol t'' is found which is in the anchor set. The state for which t'' is acceptable is q_i . Then, the error is corrected by inserting $f(q_1) \dots f(q_{i-1})$ into the input stream to the left of t'' while adjusting the original stack. Finally, normal parsing is resumed.

4. PERFORMANCE

The time and space requirements of a parser can vary widely. Seemingly small details can make huge differences. In this chapter, I bring out some of the performance issues and then compare DEER and YACC generated parsers recognizing PASCAL.

4.1. Performance Details

Conventional wisdom for software tuning is to build a system, measure it, and then work on the areas which can yield the largest payoffs. I have used this approach. The design and implementation of DEER has been heavily biased toward fast parsing. Often, clever data structures reduce both time and space requirements; however, when there has been a conflict, I have chosen to trade off some extra space for higher speed.

Figure 4 gives the static frequency distribution of directly executable code to parse PASCAL. There are total of 730 such instructions.

174	la = nexterm();
108	IF_NOT() la=parErr();
92	if (!=) goto L;
88	if (!=) la=parErr();
85	goto L;
82	PU(); goto L
36	goto pppop;
...	...

Figure 4. Frequency distribution of code

The macro IF_NOT, which tests set membership, occurs very frequently and should be fast and compact. One of the original implementations expanded this macro into about 5 machine instructions. The current version, expands the test into one instruction. This saves about 1000 bytes for PASCAL (2 bytes per instruction * 5 instructions per test * 108 tests). There are similar speed advantages to the one instruction implementation.

The director set representation is crucial to data space efficiency. There are roughly 64 director sets and 64 symbols for PASCAL. The naive implementation would require about 4096 bytes. Bit packing reduces this to 512 bytes.

We keep the lookahead token in a register since it is accessed so often. A register is also used for the base address of the data structure that the IF_NOT macro references.

The static frequency distribution of code gives us no clue as to how often these statements are executed. It turns out that director set membership is heavily used. For the input program described in the next section, the IF_NOT macro is used 37,267 times and the PU macro is used 19,590 times.

There are a number of other examples where careful tuning can yield substantial time and efficiency payoffs. These include choosing registers for heavily used objects, such as the lookahead symbol. The stack pointer is also placed in a register.

4.2. Comparison

This section compares the time and space requirements of the parsers. All measurements were carried out on a SUN 3/75 running SUN UNIX 3.2. The parsers, which are written in C, were compiled with the optimize flag (-O). The call graph execution profiler *gprof* [Graham1982b] and *time* provided the speed measurements. The *size* command provided the space information for *text*, (the executable code), *data*, (the initialized data), and *bss* (the uninitialized data, zero fill on demand). The input used was the distributed SYNPUT pascal program. The file which is 105,813 bytes long, consists of 16,170 tokens. The token distribution of this input program is given in Figure 5.

5638	Identifiers
1869	;
965	:=
834	,
740) (
400-523	Int : ^.
200-368	END BEGIN String THEN IF] [=
70-182	+ NIL DO <> ELSE VAR
40-61	- PROCEDURE WITH WHILE
...	...

Figure 5. Lexical classification of input

The DEER parser has automatic error recovery; the YACC parser has no error recovery. Figure 6 compares time and space requirements of the parsers for the input. (The link editor rounds up sizes to the next 2k byte page boundary).

Parser	Total time	Parse time	Space			
			text	data	bss	total
DEER	2.0	0.32	40960	24576	6148	71684
YACC	3.2	1.24	24576	24576	5884	55036

Figure 6. Time and space requirements

The *gprof* tool extracted the parsing time from the overall time. DEER parses about four times faster than YACC. This speed advantage plus error recovery costs about 25% more space (71K :: 55K).

5. GRAMMARS

The DEER parser generator requires an LL(1) grammar, but many existing grammars are LALR(1). This section will make the following points:

- (1) There is a need for a generator that produces fast, error recovering parsers using LALR(1) grammars. (We are currently working on this).
- (2) When designing new languages, there are good reasons to use only LL(1) parsable constructs.
- (3) Most LALR(1) grammars can be transformed into equivalent LL(1) grammars.

There are plenty of existing LALR(1) grammars and it does not make sense to transform these into LL(1) grammars just to have automatic error recovery. However, it would be desirable to have a generator that could use these existing grammars and produce fast, error recovering parsers. It could be based on an existing generator such as PGS or YACC. Corbett's [1985] Bison, which is similar to YACC, holds promise as a starting base because of its efficiency and clarity; furthermore, it is in the public domain. The major question is whether it would be easier to add error recovery to YACC or Bison, or make PGS faster. Pennello's [1986] work on making LALR parsing very fast should be considered before such a project is undertaken.

The trend has been for designers to specify more complex and more powerful languages. It is important to avoid using constructs that are difficult for humans to work with. Hoare [1981] recommends single-pass top-down recursive descent both as a implementation method and as design principle for a programming language. He says we want programs to be read by *people* and people prefer to read things once in a single pass. He concludes that there are two ways of constructing a software design: one way is make it so simple that there are *obviously* no deficiencies and the other way it to make it so complicated that there are no *obvious* deficiencies. The first is far more difficult. Wirth [1985] has similar views. He believes that the real cost of an overly complex language is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them use them efficiently. For his most recent project, he choose the simple recursive descent top-down method which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen. Philip Machanick [1986] contends that the restrictions of LL parsers discourage the adoption of language constructs difficult for the human reader to comprehend. As a grammar becomes more and more powerful, its sentences become harder and harder to understand.

There are just a few constructs that are LALR, but not LL. *Left recursion*, which is not allowed in LL grammars, is often used in an LALR grammar to specify iteration, for example:

$$\text{idlist} \rightarrow \text{idlist, identifier} \mid \text{identifier}$$

An equivalent and more transparent extended BNF grammar acceptable to DEER is:

$$\text{idlist} \rightarrow \text{identifier (, identifier)*}$$

In LALR grammars, arithmetic expressions are typically specified using left recursion as shown with the following 3 lines.

$$\begin{array}{lll} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

The equivalent, but less transparent LL grammar is given in the next 5 lines.

$$\begin{array}{lll} E & \rightarrow & T E' \\ E' & \rightarrow & + T E' \mid \epsilon \\ T & \rightarrow & F T' \\ T' & \rightarrow & * F T' \mid \epsilon \\ F & \rightarrow & (E) \mid \text{id} \end{array}$$

In the following example, known as the *dangling else*, a parser needs to decide to which *if* the *else* belongs.

```
if expression
if expression
    statement
else statement
```

The grammar is ambiguous because two different parse trees can be built. With a more powerful class of grammar (such as LALR), the ambiguity can be removed by rewriting the grammar. However, the resulting grammar is unwieldy because it contains repetitions of the original simpler grammar. The preferred approach for both LL and LALR grammars is to use an ambiguous grammar, and some mechanism that allows the ambiguity to be resolved in a natural way.

Grammars that contain alternates which can derive a common prefix are not LL. This is because it is not possible to decide which alternate to choose (unless the lookahead is extended and the common prefix is of a fixed length). The grammar can be made LL by a technique called *left-factoring*. In a simple case, such as:

```
statement    →    identifier parameter_list | identifier := expression
```

left-factoring is straightforward:

```
statement    →    identifier stmt_follow
stmt_follow  →    parameter_list | := expression
```

6. CONCLUSIONS

It is possible to produce parsers that are both fast and that have error recovery. Based only on the grammar, the parser is guaranteed to recover from any syntactic input error and will output a correct structure tree. In terms of efficiency, user friendliness and maintainability, the generated parser contributes significantly to the quality of software containing it. These design goals have been met with only a very modest space cost over parsers that have no error recovery.

7. ACKNOWLEDGEMENTS

Professor William Waite challenged me with this project and contributed with many ideas and improvements.

This work was supported in part by the National Bureau of Standards 70NANB5H0506, the Army Research Office DAAL 03-86-K-0100, and the Office of Naval Research N00014-86-K-0204.

8. REFERENCES

- [Corbett1985] Corbett, R. P., "Static Semantics and Compiler Error Recovery", UCB/Computer Science Dpt. 85/251, Berkeley, June 1985.
- [Dencker1985] Dencker, P., *User Description of the Parser Generating System PGS*, Institut fur Informatik Universitat karlsruhe, 1985.
- [Fischer1980] Fischer, C. N., D. R. Milton and S. B. Quiring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions", *Acta Informatica 13* (1980), University of Wisconsin-Madison.
- [Graham1982a] Graham, S. L., C. B. Haley and W. N. Joy, "Practical LR Error Recovery", *SIGPLAN Notices*, 1982.

- [Graham1982b] Graham, S. L., P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *UNIX PROGRAMMER'S MANUAL 4.2BSD, Vol 2C*, 1982.
- [Gray1985] Gray, R. W., "Comparing Semantic Analysis Efficiency of a GAG Generated Compiler vs Hand Written Compilers", ECE690 Report, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, Dec. 1985.
- [Gray1987] Gray, R. W., *Generating Fast, Error Recovering Parsers*, University of Colorado, Computer Science Dept., Boulder, CO, Apr. 1987. M.S. Thesis.
- [Gries1976] Gries, D., "ERROR RECOVERY and CORRECTION - An Introduction to the Literature", *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.
- [Hoare1981] Hoare, C. A. R., "The Emperor's Old Clothes", *Comm. of the ACM* 24, 2 (Feb. 1981).
- [Horning1976] Horning, J. J., "What the Compiler Should Tell the User", *Compiler Construction - An Advanced Course Edited 2nd Ed*, 1976.
- [Johnson1979] Johnson, S. C., "YACC- Yet Another Compiler Compiler", *UNIX PROGRAMMER'S MANUAL Seventh Edition, Vol 2B*, Jan 1979.
- [Machanick1986] Machanick, P., "Are LR Parsers Too Powerful?", *SIGPLAN Notices* 21, 6 (June 1986).
- [Pennello1986] Pennello, "Very Fast LR Parsing", *SIGPLAN Notices* 21, 7 (July 1986).
- [Rohrich1980] Rohrich, J., "Methods for the Automatic Construction of Error Correcting Parsers", *Acta Informatica*, 1980.
- [Waite1983] Waite, W. M. and G. Goos, *Compiler Construction*, Springer-Verlag, 1983.
- [Wirth1985] Wirth, N., "From Programming Language Design to Computer Construction", *Comm. of the ACM* 28, 2 (Feb. 1985).

Cross-Module Optimizations: Its Implementation and Benefits

Mark I. Himmelstein, Fred C. Chow, Kevin Enderby

MIPS Computer Systems

930 Arques Ave.

Sunnyvale Ca. 94086-3650

408-720-1700

decwrl!mips!{himel,fred,enderby}

Abstract

Most UNIX[†] compilers today provide module (or compilation unit) level optimizations at best, but few provide cross-module optimizations. Partitioning code into separately compiled modules deprives the typical compiler of complete program information when it performs global optimizations. The MIPS compiler solves this problem by allowing users to link intermediate code (Ucode) objects in the same way they link machine-code objects. Our Ucode optimizer can then exploit the additional cross-module information in the linked Ucode objects, providing enhanced optimizations. To link programs at the intermediate code level, we extended the same strategies used in linking at the object code level. The UNIX compiler user can take advantage of cross-module optimizations with small and straightforward extensions at the user interface. Linking programs at the intermediate code level allows users to take maximum advantage of the optimizing capability of modern compilers. Our measured results have confirmed the effectiveness of cross-module optimizations in maximizing program performance.

1. Introduction

In modern software development, modular design and structured programming are essential for rendering programs manageable. A key method for the structured organization of programs is the separation of program code into multiple source files, called modules or compilation units. Storing program code in different files facilitates the accessing, editing, sharing and maintenance of the sources. It also eases compilation overhead because only affected modules need to be re-compiled when some code is changed. Separate compilation is desirable when several programmers work on the same program and mandatory when different parts of a program are written in different languages. It is the function of the linker/loader to identify and resolve external references and common symbols when the program is finally loaded for execution.

Separate modules help programmers, but create problems for the optimizing compiler. Optimizers perform best when all of the information regarding a piece of code is available. Under separate compilation, the optimizers do not have complete information because interacting parts of the code exist in separate files. Most UNIX compilers do not provide cross-module optimizations, as compilation tools have not caught up with programming trends. Although experimental programming environments that globally record and use interprocedural information exist,¹ the MIPS compiler suite² is the first commercially available optimizing compiler that offers optimization across module and language boundaries. As a result of this cross-module optimization capability, the programmer need not to worry about reducing the quality of the compiled code when he structures his program into separate modules.

This paper addresses the approach to complete program optimization used in the MIPS compiler suite. We set out to provide more information to the optimizer without inhibiting software design concepts. Then, we enhanced our optimizer to apply some of the same concepts at the module level used at the procedure level. In addition, we implemented some optimizations applicable mostly at the module level.

[†] UNIX is a registered trademark of AT&T

We have structured our discussion into five sections. Section 2 discusses the overall compiler architecture, emphasizing the ease with which cross-module optimization support fits into the compiler system. Section 3 provides an overview of the optimizations performed on the intermediate code, and discusses their benefit from Ucode linking. Section 4 describes the essential mechanisms involved in linking the MIPS intermediate code. Section 5 gives information about the user interface when the cross-module optimization facility is invoked. Section 6 gives some program statistics showing the effects of cross-module optimization on code running on the MIPS R2000[‡] processor.³

2. Compiler overview

MIPS supports many different programming languages – C, Pascal, FORTRAN, ADA, PL/I and COBOL. The MIPS compiler system uses a common back-end to translate the intermediate code into machine code. The key is the use of a common intermediate representation for representing code from different languages. The intermediate language used at MIPS is derived from Stanford Ucode.⁴ Each language front-end translates a source file into an intermediate object file, called a Ucode object. Each Ucode object file contains two separate sections: the Ucode section contains Ucode instructions representing the program code, and the symbol table section contains symbol table information for use by the loader and debugger.

The common back-end components in the MIPS Compiler System are the Ucode optimizer (Uopt), the code generator (Ugen), and the assembler (As1). Uopt performs common global optimizations and register allocation on intermediate code. Ugen does local optimizations and translates Ucode to MIPS assembler language. The assembler does peep-hole optimizations, pipeline scheduling and produces the MIPS machine language object file. Machine code objects can be placed in libraries or linked directly into a final executable. Figure 1 shows the flow of the compiler.

When users request cross-module optimization, the compiler uses two extra phases: the Ucode linker (Uld) and the procedure integrator (Umerge). Uld links ucode objects and libraries together resulting in a Ucode object representing many modules. Umerge selectively expands procedure calls inline based on user input or its own heuristics and produces a new Ucode object. The compiler then sends this new Ucode object into the normal back-end compilation stream starting with the optimizer. To complete the compilation, the linker links the resulting machine-code object with other machine-code objects that have been separately compiled. This implies that cross-module optimization and compilation can be applied to just a part of the program structure. This allows linking in binaries or libraries that originate in assembler-code form or only come in machine-code form. We believe that this also provides an extra level of convenience to the user. Figure 2 shows the flow of the compiler with cross-module optimizations invoked.

3. Benefits

As is evident from the preceding discussion, our ability to perform cross-module optimization stems from the capability to link program files at the intermediate code level. We describe how the global optimizing transformations are applied to help understand how various optimizations benefit from the more complete program information available after intermediate code linking.

The Ucode optimizer performs many common global optimizations, including code motion, common subexpression elimination, copy propagation, strength reduction and redundant store elimination.⁵ These optimizations are applied globally within each procedure body, but are hindered by the effects of procedure calls.⁶ If the called procedures are not visible, the optimizer must assume the worst-case side effects of the calls. For example, it must assume a call will use and change all global variables and reference parameters. This often disables many useful optimizations. Intermediate code linking provides the called procedure's body in the same Ucode object as the caller, allowing the optimizer to limit side-effect assumptions.

[‡] R2000 is a registered trademark of MIPS Computer Systems, Inc.

Figure 1

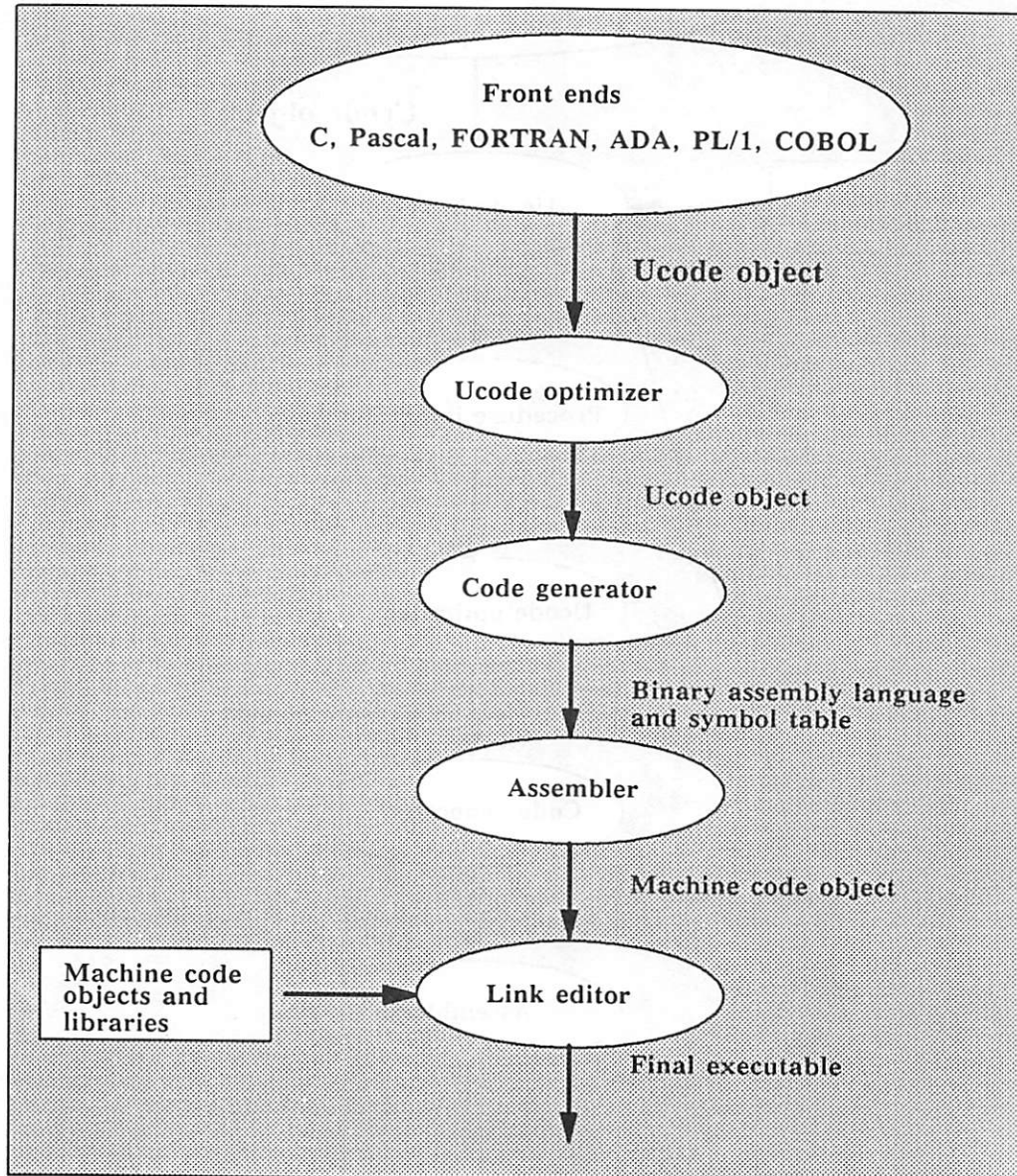
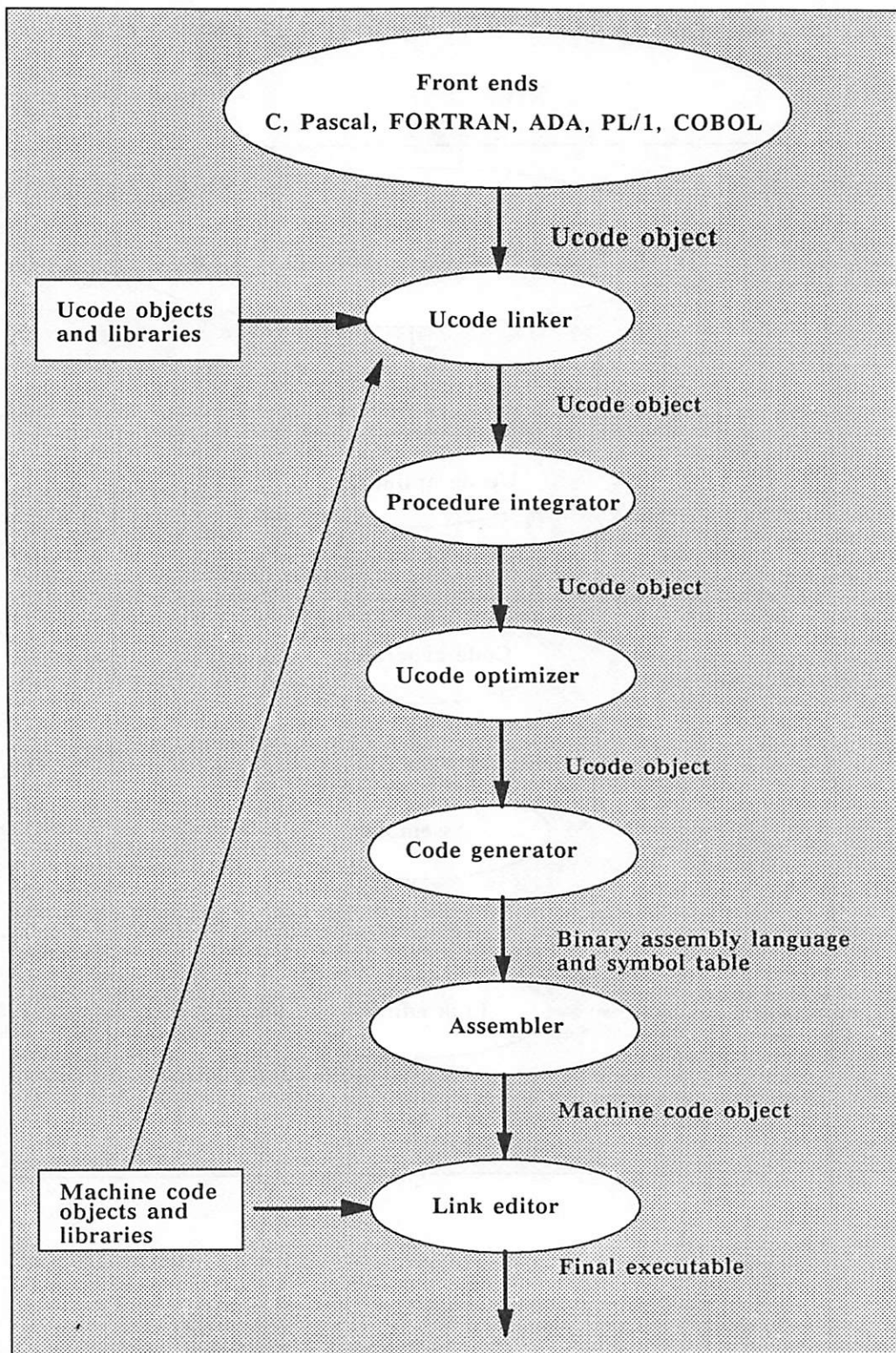


Figure 2



Another major benefit of linked intermediate code is the exposition of total usages. In linking modules together, the Ucode linker converts the external attribute of variables to internal. This guarantees to the optimizer that all potential usages of the variables are seen in the linked intermediate code, thus encouraging optimizations involving the variables. An important example concerns the aliasing of variables. When all usages are not exposed, the optimizer must assume the address of an external variable has been taken and passed around. If the address of a variable is taken, the variable will need to be included for consideration in any operation that involves aliasing. This renders expressions containing the variable invalid for common subexpression optimizations and prevents the variable from being assigned to a register.

As mentioned earlier, a procedure integration phase runs before the optimization phase to selectively inline procedure bodies at their calls based on space/time trade-offs. The effects of the inline expansion of calls have been studied and analyzed.^{7,8,9} Apart from saving the cost of procedure calls, procedure integration also enhances optimizations due to the enlarged context in the caller and the opportunity to customize the procedure body to the caller's environment. The inlining of procedure calls is possible only when the bodies of the called procedures are visible. Thus, the operation of the procedure integrator could be severely curtailed without any prior linking at the intermediate code level.

Register allocation constitutes the final phase of the Ucode optimizer. By globally analyzing usage patterns, the register allocator decides which variables best reside in registers.¹⁰ Register allocation is performed globally on a per procedure basis, but the register usage information for each procedure can be propagated to its callers. The procedure integration phase specifically re-arranges the order of procedures in the linked Ucode file to correspond to the depth-first traversal of the program call graph. Thus, procedures appear before their callers in the Ucode file. By propagating register usage information upwards via the call graph, the register allocator can avoid re-uses of the registers already used by the callees, in effect performing inter-procedural register allocation.¹¹ Linking programs on the intermediate code level ensures that all procedure bodies are exposed, enables inter-procedural register allocation to be performed effectively and thus allows the MIPS compilers to make full use of the large and uniform register set provided by the R2000 processor.

4. Ucode linking

When we included intermediate code linking in our compiler strategy, we wanted to profit from the technology already invested in our existing linker in merging and resolving symbols and accessing libraries. In fact, the Ucode linker and the machine-code linker share sources.

Uld uses the same code as the machine code linker to merge external (global) symbols so that the resulting object contains only one symbol defining (or undefining) an external name. The Ucode linker must go a step further because our compiler encodes Ucode symbol table references in unique integers, or dense numbers, for each compilation unit so that the compiler phases do not need to carry around strings in the Ucode or assembly instruction streams. Some of the dense numbers refer to externals, so the Ucode linker must merge these numbers so that a single dense number, as well as one symbol, represents any external name.

The Ucode linker must also resolve the local dense number references in Ucode instructions because the dense numbers are unique within one Ucode object but not across Ucode objects. The linker rennumbers the references so that local dense numbers in the linked Ucode object do not overlap.

As discussed above, Uld helps the optimizer by converting global symbols to local symbols whenever possible. This "localization" encourages more optimizations because local symbols do not have hidden uses. The Ucode linker needs complete linking information to accomplish localization. In addition to the Ucode objects, the Ucode linker needs to know the machine-code objects that will eventually be linked in so it can find out if any machine-code objects reference global symbols. The Ucode linker turns global symbols into local symbols when no machine-code object references them.

In order to understand “localization” we must note that in the cross-module optimization facility, linking occurs twice: once to link Ucode objects together (eventually resulting in a machine code object) and a second time to link the machine-code object, resulting from the linked Ucode object, to other machine-code objects. As an example of when the linker can not localize, consider machine-code object `crt0.o` (standard UNIX runtime startup). The C compiler links `crt0.o` as the first module of all C programs. `Crt0.o` performs startup operations (like setting up the program arguments) and then calls the user program. In Unix C programs, the user program must have a procedure called “main” which `crt0.o` can call. If in the first link, `Uld` converted “main” to a local, `crt0.o`’s reference to “main” would not be resolved in the second link.

5. Usage

Users can take advantage of cross-module optimization by linking together Ucode object files the same way they link machine-code object files. Our compiler driver and the *make* utility understand how to compile and process Ucode objects, resulting in little change in the user’s environment. The following paragraphs will show the C compiler’s user interface with and without cross-module optimization. Let `foo.c`, `bar.c` and `rab.s` be three files comprising the program `foo`, where `foo.c` and `bar.c` are C source files and `rab.s` is an assembler source file.

There are two common methods to compile these files with optimization. Both are shown in each of the following two figures.

```
first method:
    cc -O2 -o foo foo.c bar.c rab.s

second method:
    cc -O2 -c foo.c
    cc -O2 -c bar.c
    as -o rab.o rab.s
    cc -O2 -o foo foo.o bar.o rab.o
```

Figure 3

```
first method:
    cc -O3 -o foo foo.c bar.c rab.s

second method:
    cc -j foo.c
    cc -j bar.c
    as -o rab.o rab.s
    cc -O3 -o foo foo.u bar.u rab.o
```

Figure 4

Consider the methods in figure 3 for `-O2` (single-module optimization). The first method compiles all the sources and links them to make the executable `foo`. The second method produces relocatable machine-code objects (`foo.o`, `bar.o` and `rab.o`) and then links those objects to make `foo`. The second method allows users to only recompile sources that change whereas the first method always recompiles everything (an important difference when there are many source files). In both methods the user must re-link the objects after any compilation.

Now consider the analogous methods for `-O3` (cross-module optimizations) in figure 4. The first method hides the complexity from the user. The second method requires the user specify “-j” instead of “-c”, and that the user understand that Ucode objects (`foo.u` and `bar.u`) are analogous to machine-code objects (`foo.o` and `bar.o`). Note that the “-O3” flag in the second method occurs in the link line to tell the driver to Ucode link the Ucode objects and pass the linked Ucode object into the compilation stream. Also note that `rab.o` must be on the Ucode link line.

The UNIX *make* utility constitutes another important interface. The following shows a standard makefile to make “foo” as in the above example:

```
# the following three lines are macros
OBJECTS=foo.o bar.o rab.o
LDFLAGS=
CFLAGS=-O2

# the following says that foo is dependent on changes to the objects
#      and that if they do change, then execute the "cc" line to
```

```
#      update foo.
foo: $(OBJECTS)
      cc $(LDFLAGS) -o foo $(OBJECTS)
```

The *make* utility has implicit rules for generating machine code objects (".o") files from C sources and Assembler sources:

The existing implicit rule:

```
# CC is a built in macro which is set to "cc" by default
# $< is a macro for the file that caused the recompile (e.g. foo.c)
      $(CC) $(CFLAGS) -c $<

# AS is a built in macro which is set to "as" by default
# ASFLAGS is like CFLAGS, and set to nothing by default
# $@ is a macro for the file that should be created (e.g. rab.o)
      $(AS) $(ASFLAGS) -o $@ $<
```

Ucode loading simply requires an extra implicit rule and some changes to the makefile:

The new implicit rule:

```
# note the "-j" instead of the "-c" above -- this rule makes .u
#      from source files.
      $(CC) $(CFLAGS) -j $<
```

The new makefile:

```
# the following four lines are macros
OBJECTS=foo.o bar.o rab.o
LDFLAGS=
UOBJECTS=foo.u bar.u rab.o
ULDLAGS=-O3

# the following says that foo is dependent on changes to the objects
#      and that if they do change, then execute the "cc" line to
#      update foo.
foo: $(OBJECTS)
      cc $(LDFLAGS) -o foo $(OBJECTS)

# this rule is the same as above except it says foo.O3 is
#      dependent on the Ucode objects.
foo.O3: $(UOBJECTS)
      cc $(ULDLAGS) -o foo.O3 $(UOBJECTS)
```

The new implicit rule logically adds the ability to make Ucode objects. The new makefile shows how similar the rules and commands are to make a cross-module optimized version of foo. A more general makefile can support both with the same rule:

```
# the following three lines are macros
OSUFFIX=o
OBJECTS=foo.$(OSUFFIX) bar.$(OSUFFIX) rab.o
LDFLAGS=

# the following says that foo is dependent on changes to the objects
```



```
#      and that if they do changes, then execute the "cc" line to
#      update foo:
foo: $(OBJECTS)
      cc $(LDFLAGS) -o foo $(OBJECTS)
```

The only difference in the makefile is the parameterization of the suffix for the objects. By default, this makefile will act as before, but the user can specify two command-line macro arguments to *make* (`LDFLAGS=-O3` and `OSUFFIX=u` -- command-line arguments override the definitions in the makefile), and *make* will produce a cross-module optimized version of *foo*. The MIPS Language Programmer Guide provides more information.¹²

6. Measurements

In this section, we compare the performance of programs compiled with standard (`-O2`) optimization versus cross-module optimization (`-O3`). Two separate methods are used to gauge program performance. The first method times the program using the UNIX time command on the MIPS M/800 system running with the R2000 CPU under the UMIPS Operating System.[†] This is not a very accurate measurement of the compiler performance, because the actual running time is influenced by the hardware configurations, especially the amount of cache used in the system. These numbers are shown in the first row of Table 1. The second method is to use the MIPS instruction tracing facility *pixie*, which is provided as part of the profiling tools.¹² This tool generates an accurate statistical analysis of the program execution, giving full details about instruction counts and register usage. The total instruction counts are shown in the second row of Table 1.

The current production implementation of `-O3` compilation does not enable procedure inlining, and uses machine-code libraries rather than Ucode libraries. With an ideal memory system, inlining before optimization clearly reduces the number of instructions executed, and therefore reduces the execution time. But with currently feasible cache sizes, the increase in code size can degrade the instruction cache hit rate enough to negate the benefit of executing fewer instructions. Given that inlining and the use of Ucode libraries substantially increase the compilation time, we decided the compile-time cost exceeds the execution-time benefits. We expect that improvements in the memory system and better compiler heuristics will alter the tradeoffs.

Our experience shows that, of all the benefits we discussed in Section 3, the improvements attributed to inter-procedural register allocation far exceed the others, and are always visible when cross-module optimization is invoked. Thus, our analysis will focus on the register allocation. The R2000 is a load/store machine; thus any access to a memory location requires either a load or a store instruction. Register allocation strives to keep as many variables in registers as possible so as to minimize the more expensive memory accesses. It also tries to minimize the movement of register contents by keeping variables in registers as long as possible and preventing unnecessary saves and restores. Thus, a good measurement of register allocation effectiveness is the number of load and store instructions executed. These numbers are provided in Table 1, together with the program running times.

The first program is the standard Dhrystone benchmark widely used to measure processor and compiler performances.¹³ The second program, a 75000-line advance router, is a CAD program belonging to Racal-Redac's Visula software suite. The router program is written in C and organized into 350 modules. The sheer size of this program tests the effectiveness of cross-module optimization.

The results of these two benchmarks displayed in Table 1 confirm the effectiveness of cross-module optimization. The instruction counts shown do not include the degradation due to data not residing in the data cache. This accounts for the difference in improvements measured according to running time and instruction count. The improvement of `-O3` over `-O2` compilation exhibited in the table is quite typical. Our experience indicates that cross-module

[†] M/800 and UMIPS are registered trademarks of MIPS Computer Systems, Inc.

linking and optimization can reduce instruction counts by 2% to 10%.

	Dhrystone			Router		
	-O2	-O3	% improved	-O2	-O3	% improved
Running time (sec.)	2.70	2.47	8.5%	194	168	13.5%
Instruction count (Million)	30.41	29.26	3.8%	1538	1420	7.7%
Number of loads (Million)	7.10	6.40	10%	328	298	9.1%
Number of stores (Million)	3.20	2.70	16%	153	116	24.2%

Table 1

7. Conclusion

Our experience has shown that program linking at the intermediate code level can overcome the hindrance to advanced compiler optimizations created by the modularization of program sources. Our implementation is based on pure extensions of tools already existing in our system, and we have exploited the modularity and sound overall design of our compiler system. By adhering to UNIX conventions, the impact to the compiler user can be minimized. Actual performance results also confirmed the benefits that can be brought about by cross-module optimizations.

8. Acknowledgments

We would like to thank all the members of the MIPS compiler group for their help in designing, implementing and testing cross-module optimizations. Thanks to Allan Cantos, Steve Correll, Steve Hanson, John Ho, Earl Killian, Bettina Le Veille, Keith Mortensen, and Larry Weber.

1. Keith D. Cooper, Ken Kennedy, and Linda Torczon, "The Impact of Interprocedural Analysis and Optimization in the Rn Programming Environment," *Transactions on Programming Languages and Systems* 8(4) pp. 491-523 ACM, (October, 1986).
2. Fred Chow, Mark Himelstein, Earl Killian, and Larry Weber, "Engineering a RISC Compiler System," *Proceedings COMPCON*, IEEE, (March 4-6, 1986).
3. John Moussouris, Les Crudele, Dan Freitas, Craig Hansen, Ed Hudson, Steve Przybylski, Tom Riordan, and Chris Rowen, "A CMOS RISC Processor with Integrated System Functions," *Proceedings COMPCON*, IEEE, (March 4-6, 1986).
4. Peter Nye and Fred Chow, "A Transporter's Guide to the Stanford U-Code Compiler System," *Technical Report*, Computer Systems Laboratory, Stanford University, (June 1983).
5. Fred Chow, "A Portable Machine-Independent Global Optimizer - Design and Measurements," *Ph.D. Thesis and Technical Report 83-254*, Computer Systems Laboratory, Stanford University, (Dec 1983).
6. John P. Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables," *Proceedings, ACM Symposium on Principles of Programming Languages*, pp. 29-41 (Jan 1979).
7. F. E. Allen, et al., "The Experimental Compiling System," *IBM J. Res. Develop.* 4(6) pp. 695-715 (Nov 1980).
8. Robert W. Scheifler, "An Analysis of Inline Substitution for a Structured Programming Language," *Communications of the ACM* 20(9) pp. 647-654 (Sep, 1977).

-
9. Christopher A. Huson, "An In-line Subroutine Expander for Parafrase," *Technical Report UIUCDCS-R-82-1118*, Department of Computer Science, University of Illinois at Urbana-Champaign, (Dec, 1982).
 10. Fred Chow and John Hennessy, "Register Allocation by Priority-based Coloring," *Proceedings SIGPLAN*, ACM, (June 18-22, 1984).
 11. David W. Wall, "Global Register Allocation at Link Time," *Proceedings SIGPLAN Compiler Construction*, pp. 264-275 ACM, (June 23-27, 1986).
 12. MIPS Computer Systems, Inc., *MIPS Language Programmer's Guide* 1986.
 13. R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM* 27(10) pp. 1013-1030 (Oct, 1984).

RPCC - A Stub Compiler for Sun RPC

Irving Reid

Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan
Canada S7N 0W0
Usenet: reid@sask.UUCP

Abstract

A new system is described which compiles stubs for the Sun RPC system directly from annotated C programs. The specification, implementation, and usage of RPCC are described. RPCC is compared to some existing stub compilers, most notably *rpcgen*, the stub compiler provided by Sun Microsystems.

1. Introduction

Remote Procedure Call (RPC) is a primitive for handling communication between parts of a program running as separate processes. In order to simplify the interaction between the processes, RPC restricts communication to the same semantics as a local procedure call. The calling program invokes the called procedure and then waits until the called procedure returns its result before it continues execution. Communication is through the passed parameters and returned values. In general, the process which calls the remote procedure is the *client*, and the process which receives the call and performs the action is the *server*.

RPC is implemented as a series of local procedure calls at the client which establish a connection, package up the parameters, and send the call to the server. At the server end, the parameters are unpacked and handed to the desired procedure; the returned values are then packaged up and sent back. The returned values are unpacked at the client and handed back to the calling process as the result of the call. Note that the processes may, but need not, be running on different machines; RPC is just as useful for inter-process communication on a single machine as it is for network communication.

1.1. Support for RPC

Different machines can, and often do, have different representations for data in memory. For example, Suns and Vaxen do not store the bytes within an integer in the same order. Floating point numbers create even greater problems. If an RPC system is to be useful on a network connecting machines with different internal representations, some means must be provided to convert data from one representation to another. A number of solutions for this problem have been proposed. It is possible for every processor to keep track of the internal representation of every other, and convert the data before it is sent. This will be impossible for broadcast messages. The receiving processor could convert the data into its own format. This is more practical, but it still requires that each machine be able to convert data from every other machine's representation.

A simpler solution is to force every processor to convert the data into one standard format before it is sent onto the network. Each processor would only need to know how to convert network format data to and from its own internal representation. While this reduces software complexity, it could increase

overhead. Two processors with identical internal representations will be forced to translate their data to and from the network format, even though it is not strictly necessary. Hybrid solutions have been proposed where a specific internal representation is chosen for a particular connection. This would allow similar processors to communicate in native format, while different processors would work in some agreed-upon network format. This would still not work with broadcast messages.

The other issue any RPC system must address is naming. The caller must be able to specify which procedure is to be run. More sophisticated systems allow the caller to name the host machine (if it is not a broadcast call), server on that machine, version of the server, and procedure within that server. This makes naming much more flexible. Version identifiers provide support for development of new versions of servers without disabling existing versions; likewise, explicit host naming allows a process to address a call to the same server on any chosen machine.

The Sun RPC implementation provides support for Remote Procedure Call in the 4.2BSD Unix(tm.) networking environment. Data on the network are kept in a fixed internal representation, the Sun External Data Representation (XDR) [6]. Library functions are provided to assist the programmer in converting data to and from the XDR, and for establishing the underlying network connections. The programmer is responsible for providing XDR conversion routines for non-standard data structures and for writing stubs to do the actual remote call. Host addressing is done either through network names or Internet addresses. Servers are given unique program and version numbers, and procedures within each server are also numbered.

1.2. Stub Compilers

Stub compilers, and RPCC in particular, automatically generate the code which addresses the remote procedure and converts data into and out of network format. The “stubs” themselves are functions which make the remote call completely transparent. Because RPCC generates both the server and client, the client stub can simply be given the server name and program, version and procedure numbers. The client stub puts all the parameters into a single structure and invokes the Sun RPC library routines to perform the call. The server stub receives the call and takes the parameters out of the structure to pass them to the remote procedure.

The RPCC stub compiler automatically generates XDR functions and procedure stubs from C programs, using the Sun RPC library routines provided. Programs are annotated by putting the special comment `/*REMOTE*/` before functions which are to be called using RPC. RPCC determines the data structures being communicated and produces the necessary routines for converting these to and from the XDR; it also generates a stub procedure with the same name as the remote procedure, for the client to call, and another stub which calls the procedure at the server end. Simple escapes are provided to allow programmers to use their own XDR routines for special purposes.

Because all RPCC directives are C comments, programs using RPCC may be compiled and tested without RPC. The program can then be fed unchanged to RPCC to produce the stubs and XDR routines. The programmer need not worry about introducing errors while converting a program from ordinary to remote procedure calls.

RPCC differs from *rpcgen*[4], the stub compiler provided by Sun, in a number of ways. *Rpcgen* uses a different language, which is close enough to C that it may be confusing to the programmer. The specifications for *rpcgen* are separate from the program and may become inconsistent. *Rpcgen* produces only the XDR routines and not procedure stubs, and does not handle many valid C structures.

2. Major Design Issues

A number of problems must be solved in designing a stub compiler. First, some communication mechanism and network data representation must be provided. In this case, these are provided by Sun RPC. The remaining issues depend on the level of service desired. How transparent should a remote procedure call be? How can the compiler determine which functions may be called remotely? Which data types should be supported, and how should they be specified?

Rpcgen solves the problems of data type and function interface specification by inventing a new language, *rpcl* (for Remote Procedure Call Language). The language allows the programmer to specify data types including all simple C types, structures (though not nested structures, in the version distributed with SunOS 3.2), strings, and two special types: counted length arrays, which consist of a count of the number of elements and a pointer to the first element; and discriminated unions, which contain a variable determining how the contents of the union should be interpreted, followed by the actual union. Parameter types are specified with a C-like syntax, but a completely different syntax is used for specifying procedure and program numbers. Procedures are restricted to a single parameter and return value, and true call-by-reference (where changed reference parameters are copied back) is not supported.

Nordin's Remote Operation Calls [3] provide an alternate method of specifying remote interfaces. Comments are placed in the source file to specify the program and procedure names and the parameter types. Different comments are used depending on whether a file is exporting or importing the remote interface. An X.400 based name-server is used for both setting up communications and for storing type-checking information about remote procedures. Nordin's stub compiler handles both value and reference parameters, correctly copying back new values of reference parameters. The actual return value is not passed back; results are returned through reference parameters. It seems (from the published description) to only handle simple data types.

Where both of these methods break down is that the specification is separate from the implementation. Thus it is possible for a program which runs correctly when compiled as a single process to fail when it is compiled with the stub compiler, if the specification does not reflect the actual state of the program. The primary motivation behind RPCC is that of *source transparency*. A system which derives the remote interface directly from the C program will be more reliable, since the programmer does not have to worry about keeping a separate specification up to date. Also, it becomes possible to use the C compiler and *lint* to test program code, as long as stub compiler directives do not change the syntax of the language. This is achieved by using C comments to direct the stub compiler.

The first problem is to identify which procedures must have an RPC interface. Because C does not have any modular structure aside from files and static and extern declarations, we cannot deduce this information from module interfaces. Instead, each procedure which will be called through RPC is marked with the comment `/*REMOTE*/`. The stub compiler will parse out the parameter and return types for the function, and generate any necessary XDR functions.

Many common data structures in C are implemented using pointers to objects. Strings are the prime example of this. A pointer to a character is almost always actually a pointer to the first of a null-terminated array of characters. Because this sort of trickery cannot be reliably determined by a stub compiler, no attempt is made to second-guess the programmer. A pointer to a character is just that; if the programmer wishes to pack strings a typedef and the `/*XDR=name*/` directive must be used. Similarly, Sun RPC assigns no special meaning to null pointers; attempting to pass one will usually crash the program. If the programmer wishes to preserve null pointers, for instance to pass a null-terminated linked list, a user-defined XDR function must be used.

The C “union” allows a number of different representations to apply to a single memory location. In general, there is no way to know which representation applies at any given time. *Rpcgen* provides a *discriminated union*, in which a control variable uniquely determines the representation which should be used at run time. While a similar mechanism could be provided through RPCC directives, the complexity of such a measure precluded its implementation.

This author’s mind boggles at the semantic and implementation problems involved in carrying function pointers across machines. Mercifully, there is little call for such a facility; no attempt is made to support it.

The Sun RPC library procedures used by RPCC restrict remote procedures to a single parameter and return value, each of which may be an arbitrary data type. In order to handle general procedure calls, the RPCC client stub must take the parameters from multi-parameter functions and create a single structure to hold them all. The server stub must take these parameters back out of the structure before they are passed to the remote procedure.

3. Using RPCC

The simplest use of RPCC is to take a working program and make one function in that program into a remote procedure. The function must be in a separate file from the main (client) program, so that they can be compiled separately using *cc* after the stubs have been generated. The function to be called remotely should be immediately preceded by the special comment */*REMOTE*/*. The client program must somewhere define a character string called *HostName*, which must (at the time the stub is invoked) contain the name of the machine the server is running on. The generated program uses the default program and version numbers *RUSERSPROG* and *RUSERSVERS* from *<rpcsvc/rusers.h>*. The remote procedures are numbered in the order they are found in the source files.

As an example, we will develop a program called *radd* (Remote ADD), which is contrived to demonstrate a few features. The program consists of a main routine which calls a function *add()* of two integer parameters. *Add()* returns a pointer to a *struct add_val*, which contains the two summands and the sum; this is chosen to demonstrate XDR function generation for structures. The main routine is defined in file *main.c*, *add()* is defined in file *add.c*, and the structure is defined in file *defs.h*. Compiling this with the command

```
cc -o radd main.c add.c
```

produces a program *radd* which runs on a single machine.

The stubs are generated by running RPCC on all of the files which contain remote functions for a given server. Necessary definitions are placed in the file *X_xdr.h*, XDR conversion functions in *X_xdr.c*, client stubs in *X_client.c*, and server stubs and the server *main()* in *X_server.c*. All remote procedures must be done at once, since RPCC gives consecutive numbers to all the procedures found in one run and puts the output in fixed file names. The commands

```
rpcc add.c
cc -o client main.c X_xdr.c X_client.c
cc -o server X_server.c X_xdr.c add.c
```

will produce an RPC version of *radd* where the program *server* can be run on one machine and *client* on another. Listings of *radd*, as well as the code produced by RPCC, may be found in Appendix A.

3.1. User Defined XDR Functions

Automatic generation of XDR functions for some common data structures, such as strings, unions, and variable length arrays, is not supported by RPCC. In order to support a wider variety of data structures, RPCC allows the programmer to associate handwritten XDR functions with specific data types or structure elements. This is done using another special comment, of the form `/*XDR=name*/`. The comment is placed immediately before the name in the typedef or structure declaration. For example,

```
typedef char * /*XDR=xdr_wrapstring*/ STRING;

struct /*XDR=xdr_foo*/ foo
{
    int bar;
    int fumble;
};

struct joe
{
    int bob;
    short /*XDR=xdr_sue*/ sue;
};
```

will associate the Sun XDR function `xdr_wrapstring()` (necessary for passing string parameters) with the type `STRING`, the user-defined function `xdr_foo()` with the type `struct foo`, and `xdr_sue()` with field `sue` within `struct joe`. Anywhere RPCC would normally generate an XDR function for that type or invoke one for that type or field, the specified XDR function will be used instead.

3.2. More Sophisticated Uses

The stubs generated by RPCC are limited by the middle-level interface to the Sun RPC library. The middle-level interface does not support call-back, authentication, broadcast RPC, or use of different network protocols. All of these can be performed using lower level interfaces to the Sun library, described in Sections 3 and 4 of the Remote Procedure Call Programming Guide [5]. Some hand modification of the RPCC stubs may be necessary (for instance, removing the main program from `X_server.c`) in order to use the stubs with lower-level RPC facilities. This could be supported with compiler options in future.

4. Implementation

RPCC is implemented in C in the 4.2BSD Unix environment. Lex and Yacc were used to generate the scanner and parser, respectively, which parse the declarations from the C source and build tree data structures which represent the functions, parameters and data types for which stubs should be generated. After all the source is processed, the definition trees are traversed to generate the necessary code.

The front end from *lint* or *pcc* could have been used to parse the source; this would have provided a more general and robust parser. However, sufficient documentation for the output from these was unavailable during the time the system was being implemented. Modifying the *pcc* source was not an option because it would preclude the use of RPCC at sites without source licenses. Rather than trying to reverse-engineer the existing front ends, a simple parser was implemented which only reads the global and function definitions from files, and ignores all code and declarations inside functions. This removed all need for

expression parsing, which is the largest part of any C grammar.

Once the declarations have been parsed, RPCC takes each remote function in turn and generates any stubs necessary. Each remote procedure is given a number, through a `#define` in `X_xdr.h`. For each remote procedure `name`, the constant `RP_name` gives the unique number for that procedure. Definitions for all structures and enums, including structures created to handle multiple parameters, and external definitions for all XDR and stub functions are also written into `X_xdr.h`. The data structure trees for the function are traversed bottom-up and XDR functions are generated where needed. These functions are named `xdr_name()`, where `name` is the name of the data type or struct.

After all XDR routines are generated, the client and server stubs are created. These are placed in `X_client.c` and `X_server.c`. The client stub places multiple parameters into a structure if necessary, and then uses `callrpc()` to call the remote procedure. The server main routine registers the server stub using `registerrpc()` and then waits for a call to arrive; when a call comes in, the server stub unpacks multiple parameters if they exist, calls the remote procedure, and returns the result; the Sun library (through `registerrpc()`) ensures that the return value is packed correctly.

5. Current Limitations

The current implementation supports all simple data types: `int`, `char`, `long`, `short`, signed, unsigned, `float`, `double`, and `void`; it can also handle arbitrarily nested structures, enums, fixed length arrays, and pointers to any supported data type (provided the pointers are not null). Unions, variable length arrays, and pointers to functions are not directly supported, though the programmer can use support functions provided with Sun RPC to simplify implementation of unions and variable length arrays.

The current implementation does not support reference parameters. Modified parameters will not be copied back, so the changes will not be reflected in the host environment. Support for this would be relatively simple to implement; any parameter which contains a pointer reference would be included in a structure with the return value. The server stub would return the reference parameter along with the return value, and the client stub would unpack it into the local reference parameter, leaving the modified value in the caller's address space.

6. Future Work

Many possible extensions have been mentioned in the preceding text. Some method should be devised to handle unions. More support should be added for programmers who wish to use the lower-level RPC library routines. The system should handle reference parameters correctly.

In the long run, the Sun RPC library routines provide a very cumbersome interface, especially at lower levels. RPC programming in general would be much simpler with a system like the standard I/O library. Remote procedure calls would go to a default server, set up at program initialisation time. Other servers could be "opened" and handled using a server descriptor, much like `stdio` uses file descriptors. Passing a server descriptor to a calling stub will cause that stub to call its counterpart at the specified server. This could also be used to support callback, by making the client's server descriptor one of the parameters to the server stub.

C is a rather difficult language to handle RPC in, because it does not directly support packaging of sections of a program into separate modules with cleanly defined interfaces. RPC is much cleaner in modular languages with some support for object oriented programming, such as Eden's EPL [1] and Argus' version of Clu [2]. Perhaps C++, with its support for modules and object-oriented programming, would be better as a base language. However, until C++ comes into much wider use some C tools are still

necessary.

7. Conclusions

The Sun RPC library, running on the Berkeley Unix networking system, provides a workable interface for generating applications using Remote Procedure Call. There are a number of drawbacks, most notably the relatively high overhead of high-level RPC and the complexity of the low-level interface. This is offset by the support for heterogeneous environments and the relative simplicity of writing applications using the high-level services.

RPCC provides the programmer with a direct migration path from single process to multiple process applications by generating Remote Procedure Call stubs directly from C code. This differs from known existing C stub compilers, which require a separate specification of the remote interface. By making the RPC extensions source transparent, traditional C tools can be used to develop applications as single processes which can then be recompiled into client and server modules without modifying the source code.

8. Acknowledgements

The author is most grateful to Derek Eager and Susan Tourigny for comments on drafts of this paper. This work has been supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

9. References

1. G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe, The Eden System: A Technical Review, *IEEE Transactions on Software Engineering SE-11* (January 1985), 43-59.
2. B. Liskov, Primitives for Distributed Computing, *Proceedings of the 7th Symp. on Operating System Prin.*, 1979, 33-42.
3. B. Nordin, I. A. Macleod and T. P. Martin, Remote Operations Across a Network of Small Computers, *Proceedings of the 1986 ACM Sigsmall/PC Symposium on Small Systems*, San Francisco, Dec. 3-5, 1986.
4. *rpcgen - An RPC Protocol Compiler*, Sun Microsystems, Inc., 1986.
5. *Remote Procedure Call Programming Guide*, Sun Microsystems, Inc., 1986.
6. *External Data Representation Protocol Specification.*, Sun Microsystems, Inc., 1986..

Appendix A - sources for ‘radd’

==> main.c <==

```
/* main program for rpcc demonstration */
#include <stdio.h>
#include "defs.h"
char *HostName = "sketch";

main()
{
    struct add_val      *temp, *add();

    temp = add(2, 3);
    printf("%d + %d = %d\n", temp->p1, temp->p2, temp->sum);
    exit(0);
}
```

==> add.c <==

```
/* add function for radd - rpcc example program */
#include "defs.h"
/*REMOTE*/
struct add_val *
add(p1, p2)
int    p1, p2;
{
    static struct add_val result;

    result.p1 = p1;
    result.p2 = p2;
    result.sum = p1 + p2;
    return(&result);
}
```

==> defs.h <==

```
/* data definitions for radd - rpcc example program */
struct add_val
{
    int    p1, p2;
    int    sum;
};
```

==> X_client.c <==

```
/* Sun RPC interface automatically generated by RPCC */
#include "X_xdr.h"
extern char *HostName;

struct add_val      *add(p1, p2)
int    p1;
int    p2;
```



```

{
    struct add_val      *retval;
    struct X__0    temp;

    temp.p1 = p1;
    temp.p2 = p2;
    if (callrpc(HostName, RUSERSPROG, RUSERSVERS, RP_add,
                xdr_X__0, &temp, xdr_X__1, &retval) != 0)
    {
        write(2, "remote call to add failed\n", 26);
        exit(1);
    }
    return(retval);
}

==> X_server.c <==
/* Sun RPC interface automatically generated by RPCC */
#include "X_xdr.h"

char *
XX_add(args)
struct X__0 *args;
{
    static struct add_val      *retval;
    retval = add(args->p1, args->p2);
    return((char *)&retval);
}

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RP_add,
                XX_add, xdr_X__0, xdr_X__1);
    svc_run();
    write(2, "Error: svc_run returned\n", 24);
    exit(1);
}

==> X_xdr.c <==
/* Sun RPC interface automatically generated by RPCC */
#include "X_xdr.h"
bool_t
xdr_add_val(xdrsp, str)
XDR      *xdrsp;
struct add_val      *str;
{
    if(!xdr_int(xdrsp, &str->p1))
        return(FALSE);
    if(!xdr_int(xdrsp, &str->p2))

```

```

        return(FALSE);
    if(!xdr_int(xdrsp, &str->sum))
        return(FALSE);
    return(TRUE);
}

bool_t
xdr_X__0(xdrsp, str)
XDR *xdrsp;
struct X__0 *str;
{
    if(!xdr_int(xdrsp, &str->p1))
        return(FALSE);
    if(!xdr_int(xdrsp, &str->p2))
        return(FALSE);
    return(TRUE);
}

bool_t
xdr_X__1(xdrsp, str)
XDR *xdrsp;
struct add_val **str;
{
    if(!xdr_reference(xdrsp, str, sizeof(**str), xdr_add_val))
        return(FALSE);
    return(TRUE);
}

==> X_xdr.h <==
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
struct add_val
{
    int p1;
    int p2;
    int sum;
};
#define RP_add 1
struct add_val *add();
bool_t xdr_add_val();
struct X__0
{
    int p1;
    int p2;
};
bool_t xdr_X__0();
bool_t xdr_X__1();

```

Implementing the Reliable Data Protocol (RDP)

Craig Partridge

Harvard University/BBN Laboratories Incorporated¹

Abstract

The author examines the problems of implementing a reliable IP-based protocol, including issues such as choosing timer algorithms, effecting congestion control mechanisms, and integration with the BSD operating system. A preliminary evaluation of the protocol is presented.

1. Introduction

During the fall of 1986 the author implemented the Reliable Data Protocol (RDP), a reliable, connection-oriented, IP-based transport protocol under the 4.2 and 4.3 BSD operating systems. RDP differs from the better-known Transmission Control Protocol (TCP) [13] in that it keeps data boundaries; data is presented to the receiving end in the same data units that it was written in by the sender. Another difference is that, unless the receiver requires it, RDP does not preserve order; units may be delivered in an order different from that in which they were sent. Finally, as a result of the different external interface, RDP has different internal mechanics from other reliable protocols.

Because of these differences, RDP provides another perspective on the problems of implementing a reliable protocol. This paper explores that new perspective and gives some attention to the question of whether this new form of transport service is useful.

2. The Protocol

RDP is specified in Internet Request for Comments 908 (RFC908) as amended in a new RFC, which specifies version 2 of the protocol and is expected out soon [4,16]. The goal of its designers was to develop a simple transport protocol for bulk data transfer to be used on networks with moderate loss rates.

The protocol state diagram is shown in Figure 1, and illustrates the way the protocol works.² There are two ways to open a connection. Peers start in the closed state and use either an active or a passive open request to reach the connected (OPEN) state. The connection is full-duplex, and data is acknowledged as it is received. Either side may initiate the request to close.

RDP accepts data units from higher layers, typically applications, which are then put on the network. In this paper, blocks of data from the higher layers are called *records* and blocks of data passed to the network are called *packets*. For RDP, there is really no distinction. A record from an application is sent out as a single packet, and a packet can contain only one record.³

Multiple data packets may be in flight simultaneously. (The term *data packets* is used throughout the paper to distinguish from control packets such as SYNs, RSTs and ACKs). The limit on the number of packets in flight is called the *window size*. Unless the receiving peer

¹ This paper describes work done at Harvard University towards the author's M.Sc. degree, under the supervision of Professor Meichun Hsu. The author can be reached c/o BBN Laboratories, 10 Moulton St, Cambridge MA 02238, or via network mail at craig@sh.cs.net or craig@harvard.harvard.edu.

² The diagram is adapted from the one on page 10 of RFC908.

³ RDP is uncommon in this respect. Most reliable protocols support mechanisms for aggregating or subdividing data into more convenient packet sizes.

requests sequencing, records are delivered when their corresponding packet is received. Each data packet is given a thirty-two bit sequence number, and packets are acknowledged by sending the sequence number of the highest numbered packet received in sequence. Sequence numbers wrap around, so 0 is "higher" than $2^{32}-1$. RDP also provides an uncommon extended acknowledgement (EACK) mechanism designed to reduce unnecessary retransmission by acknowledging packets received out of sequence. An EACK acknowledges a packet by sending the sequence number of the packet received.

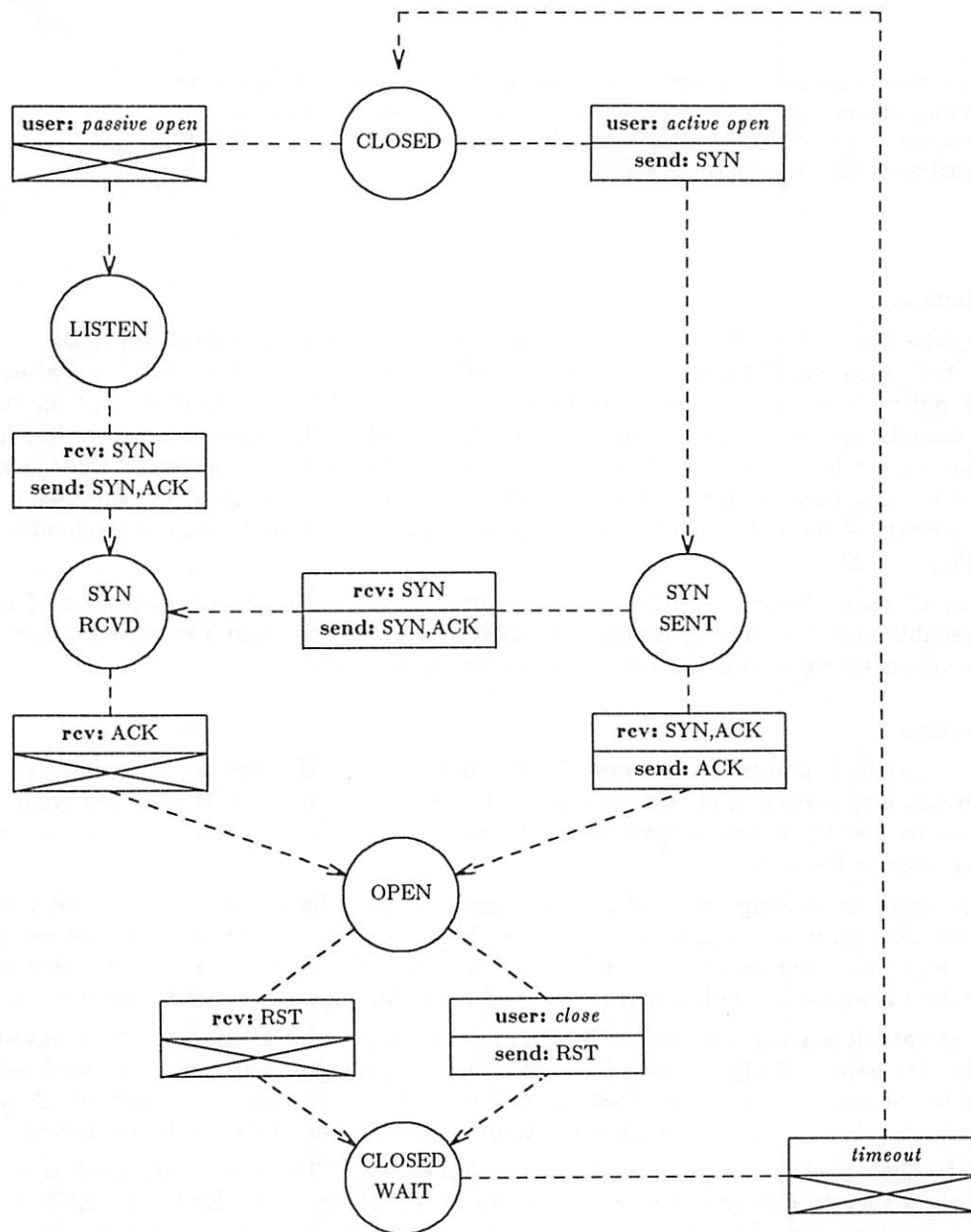


Figure 1: RDP State Diagram

Data flow is managed in two ways. After the window has been filled (i.e., the number of data packets in flight is equal to the window size) packets may be sent only after an outstanding packet has been acknowledged. EACKs are useful here because by acknowledging out-of-sequence packets, they allow new packets to be sent. The second control is that, even with

EACKs, the sequence numbers of all the outstanding packets must be within a range of twice the window size. This limit is called the *range size*. The window size is negotiated when a connection is opened and cannot be changed.

RDP promises to deliver each record reliably and only once to the destination. Duplicates are detected by reference to the sequence number. To ensure delivery, each data packet has timer associated with it, and packets are retransmitted if they are not acknowledged within an estimated round-trip time. A checksum is used to ensure that data has not been corrupted in transit. Originally, RDP used a 32-bit non-linear checksum. Testing revealed that this checksum was very expensive to compute on machines that did not use network bit order; on machines with comparable hardware but different bit order the time to compute a checksum could differ by a factor of five. The checksum has recently been changed to use the 16-bit TCP checksum.

The Internet Protocol (IP) is used as the network protocol [11]. IP identifies incoming RDP packets through a unique and well-known protocol number in the IP header and passes the packets to the RDP layer. RDP demultiplexes records to the appropriate applications using a 16-bit port number in the RDP packet header.

3. Implementation Goals

Writing protocol specifications is still an evolving art form, but specifications are generally written to completely define the protocol's external behaviour yet they say little or nothing about how that behaviour should be implemented. As a result, the implementor has considerable flexibility when designing the code. Since different design goals can lead to very different performance, it is worth taking a moment to discuss the author's goals when designing this implementation.

The primary goal was to write an implementation that transferred data as swiftly as possible, consistent with proper use of the network. The notion of "proper use" is still ill-defined, but is generally felt to embody the principle that each connection is only one of many users of a network, and should not be permitted to use the network in ways that are detrimental to other connections. Two areas of concern are unnecessary traffic, usually caused by excessive retransmissions of data packets, and congestion control, which requires that connections be able to reduce their demands on an overburdened network.

There were several secondary design goals. First, the implementation was intended to be reasonably portable across different machines running the BSD operating system. This implementation was tested on two different architectures: a Sun Workstation⁴ and a VAX-11/750.⁵

Second, the author decided to implement the entire protocol in a single round of design and coding. Historically reliable protocols have been implemented incrementally, by building progressively larger subsets. This may make sense when the protocol is complex or contains several experimental features [2]. But the risk with this approach is that the code may become a patchwork; when new subsets reveal deficiencies in earlier subsets the instinct is to patch the earlier subset instead of properly redesigning the code. While RDP has some novel features, it is not particularly complex. Implementing it in one design and coding cycle seemed likely to produce more integrated code.

This approach to coding worked out. The implementation process took about two weeks.⁶ While a certain amount of modification has been required for performance reasons, the overall structure of the code has held up well.

Finally, the author had to deal with the danger of building a fast implementation which derived its speed from using an unreasonable amount of data space in the kernel. To ensure this

⁴ Sun Workstation is a registered trademark of Sun Microsystems, Inc.

⁵ VAX is a registered trademark of Digital Equipment Corporation.

⁶ In contrast, performance testing has taken months.

did not happen, the author decided to limit the amount of per-connection state information that the kernel stored. Initially this limit was the size of a single memory buffer (*mbuf*); later this limit was revised to two *mbufs*.⁷

4. Implementation Issues

There were four major issues that had to be addressed when building the implementation. Two issues, outbound data management and inbound flow control, involved problems with the BSD kernel network architecture. The other two issues, congestion control and retransmission timer algorithms, are encountered in most reliable protocol implementations.

4.1. Outbound Data Management

Because RDP is reliable, the sending peer in a connection must keep a copy of every packet sent until the packet is acknowledged. Because RDP allows packets to be acknowledged out of sequence, it must be possible to locate and delete copies of any packet swiftly, preferably in $O(1)$ time. Unfortunately, the BSD kernel network architecture makes locating packets an $O(w)$ operation, where w is the window size. Getting the desired performance required an esthetically unpleasant workaround.

In the BSD architecture, applications access transport protocols such as RDP through the use of a *socket*. A socket is simply the application's link into the protocol layer. When an application wants to access the network, it opens a socket with certain characteristics such as a transmission mode (stream or datagram) and protocol family (Internet or PUP). The kernel links the socket to the appropriate transport protocol [6,7].

To send, the application writes data to the socket. The socket layer checks to see that the transport layer is prepared to accept the data. If the transport layer is unable to accept the data, (i.e., if the connection has consumed its available buffer space) then the socket layer is responsible for blocking the application until the transport layer can accept the data. Herein lies the problem.

The socket layer and transport layer share the queue that contains all unacknowledged outbound packets; thus determining if the transport layer can accept new data is simply a matter of finding out if the size of this queue is below some limit. Unfortunately, the queue is implemented as a linked list of *mbufs*, where a single packet may span more than one *mbuf*. Access to any packet other than the first is proportional to the amount of data in the queue. This seemed too expensive for RDP.

The solution was to have the RDP code bypass the shared queue and store the outstanding packets in a ring buffer in the transport layer. This is a bit expensive in terms of state information but gives the desired $O(1)$ access time. It also means the socket layer cannot determine whether any more data can be submitted to the transport layer. However, for efficiency reasons, the shared queue actually stores its length in a counter so that the socket layer doesn't have to re-compute the length of the queue every time it gets new data. It is thus possible for the RDP code to directly manipulate these counters and cause the socket layer to block when appropriate without actually storing the data in the queue. That is what this implementation does.

4.2. Inbound Flow Control

Inbound flow control posed a problem similar to the problem with outbound data management.

When the transport layer receives inbound data from the network layer, it must demultiplex the data and pass the data to the appropriate application. In the case of RDP, this means passing the inbound record up to the socket layer. As with outbound data, managing the transfer

⁷ The change does not represent a slackening standard. It turned out that one *mbuf*, which contains only 112 bytes of space, was too small. Late in the implementation process the author found himself recomputing values in critical portions of the code because there was no space to store them.

between layers is handled through a shared queue. And, as with outbound data, there is a limit on how much data can be placed in the queue. But in this case, the limit represents the maximum amount of unread data that the socket layer is allowed to have queued up for the application. If data is arriving faster than an application can read it, the transport layer is expected to exercise dynamic flow control to keep from overfilling the queue.

Unfortunately, RDP does not support dynamic flow control. There is no mechanism to change the window size after the connection is opened. So the author needed to figure out how to deal with the socket layer's requirement that the transport layer exercise flow control? Three options were considered.

One possibility was to ignore the limits on the queue size. The inbound queue is a linked list, and thus has no real size limit. The maximum queue size is simply an informal agreement between the socket and transport layers about how much data will be allowed to be queued at any given time; the transport layer is free to ignore the limit.

The strongest argument in favor of ignoring the limit on the queue size is that many application protocols incorporate periodic resynchronization. For example, when mail is transferred using the Simple Mail Transfer Protocol (SMTP), the interaction is a series of exchanges [14]. The sender sends a piece of data (such as an address or the message text) and waits for a reply from the receiver ("address ok" or "message text received"). In other words, application protocols often implement their own flow control which can replace the absence of flow control in RDP.

But there is a class of applications, in particular the bulk transfer applications that RDP was designed to serve, in which the interaction is between a sender, transmitting data as fast as possible, and passive receiver, reading the data. Using RDP, a slow receiver paired with a fast sender will rapidly consume all the buffer space on the receiving machine if the inbound queue size limits are ignored. It is undesirable to allow a single application to consume all of a machine's buffer space, and the author concluded that the queue limits had to be observed.

Another option was to try to find some way to support flow control with RDP. RDP could be revised to support dynamic changes to the window size, but that is a complex solution, requiring more research and experimentation than the author was willing to undertake.⁸ An alternative was to use the network congestion control mechanism to implement flow control (see below), since mismatched data rates between the transport and socket layers feels similar to network congestion (somewhere in the link, something cannot keep up with the current transfer rate). But it is not clear that congestion control and flow control are so similar that the same mechanism can be used to manage both problems [15]. Finally, crude flow control could be implemented by delaying acknowledgment of new data when the inbound queue threatens to fill. This works but has a variety of unpleasant side-effects such as burdening the network with unnecessary retransmissions and artificially increasing the estimated retransmission times at the sending peer.

If flow control isn't used and it is important to limit the queue size, there is only one remaining solution: break the connection if the queue gets too large. That is what this implementation does. As a result, if an application mis-estimates how fast it can read data, it suffers the penalty of having its connection broken. Since the estimate of how fast data can be read may be a function of variables not under the application's control (such as processor load) this is not a perfect solution.

4.3. Minor Problems with the BSD System

There were a few other minor implementation problems with the BSD system that are worth mentioning.

A lot of the support routines in the BSD code assume that the protocol uses a sixteen-bit port number. While version 2 of RDP uses a sixteen-bit port numbers, version 1 used only eight

⁸ Active research is still being done on the management of TCP window sizes, and TCP has been in heavy use since the early 1980s.

bits. As a result, several standard routines had to be re-implemented in eight-bit versions for RDP. The problem is not unique to version 1 of RDP; other Internet protocols such as the Host Monitoring Protocol (HMP) [3] also use eight-bit port numbers.

In a few places, the BSD code assumes that a protocol supports only one type of service (e.g., stream, or datagram). This isn't true of RDP; it can be used to support three different types of service: stream, reliably-delivered-message, and sequential packets. A certain amount of work was required to make sure that entry points were not called three times (one for each type of service supported).

4.4. Congestion Control

Recently congestion control has become a major topic of interest in the networking community because explosive growth on the major research networks, in particular the DARPA Internet, has overburdened the existing network infrastructure [9,10]. No transport protocol implementation is considered complete without a mechanism for responding to network congestion.

On IP networks, congestion is detected by network gateways and hosts. When gateways or hosts receive IP packets faster than the packets can be processed, they are supposed to send an Internet Control Message Protocol (ICMP) Source Quench message to the sender of the offending IP packets [12]. The sender is then expected to take action to reduce the rate at which data is sent.⁹

On BSD systems, the quench message is translated into a request to the transport layers to reduce data sent to a given destination. In RDP's case, this means the implementation must locate all connections to the destination, and take actions to reduce the traffic flow. In this implementation that means reducing the sending window size to half the number of packets that are currently in flight. Shrinking the window size ensures that no new data packets will be put on the network and that the number of packets in flight will diminish to a more acceptable level.

After a period of time, if no new quench requests are received, the sending window size will slowly increase to its maximum. New quench requests will cause the window to shrink again. While this oscillation of the window size might appear inefficient, it is necessary. Except in rare situations, congestion is a transient condition. To make proper use of the network, transport protocols must be able to increase their data rate when the available network bandwidth increases. (It may help to think of the situation of encountering slow-moving heavy traffic on a highway. When the traffic clears, one tends to want to return to higher speeds).

4.5. Retransmission Timer Algorithms

RDP requires that the sending host keep a timer for every packet sent. The packet is retransmitted if an acknowledgement is not received within an estimated transit time called the retransmission timeout (RTO). The method for estimating the RTO is not specified. This implementation uses a modified version of the TCP algorithm.

In the TCP algorithm, the time between sending a packet and receiving an acknowledgement for that packet, the round-trip time or RTT, is sampled during the lifetime of a connection. The RTTs are used to compute a smoothed round-trip time, the SRTT, using the following algorithm:

$$SRTT = (\alpha \times SRTT) + ((1 - \alpha) \times RTT)$$

The SRTT is then used to compute the RTO:

$$RTO = \beta \times SRTT$$

Based on Mills' work [8], this implementation uses different values for α depending on whether the

⁹ This is one of several possible congestion control mechanisms. See [15] for some others. Note that some recent work on congestion control assumes that retransmission timers are sufficient to track congestion [5]. Such schemes work if congestion is the only cause of lost packets; this is not the case on most IP networks.

RTT is greater than or less than the SRTT.¹⁰

The behaviour of the TCP algorithm has been analyzed in detail by Zhang, and timer issues in general have been examined by Watson [17,18]. The next few paragraphs look at some of these issues and how they related to RDP.

A critical problem is estimating the RTO properly. Zhang suggests that it is impossible to correctly estimate the RTO on a network where packets are occasionally lost (as they are on IP networks). But some methods of estimating are worse than others. In general, if one wants the protocol to be fast, the RTO should be nearly the value of the SRTT. This allows dropped packets to be detected quickly. Unfortunately, this is disastrous for network utilization, because it tends to cause unnecessary retransmissions. For example, if the RTO was set to the SRTT and the SRTT was an accurate mean of the RTTs, roughly half of all retransmissions would be unnecessary. It is generally believed that the RTO should be at least twice the SRTT to prevent unnecessary retransmissions (i.e., $\beta \geq 2$).

If the RTO is twice the SRTT, then there is an interest in making sure the SRTT is correct. In particular, there is a strong urge to keep it from being a high estimate. This is difficult. There are two methods for measuring RTTs: measuring the round-trip interval from the first time a packet is sent, or measuring from the time of the most recent retransmission. Neither method works well. Estimating from the first transmission gives high RTT values when the network drops packets because the RTT for retransmitted packets includes the RTO. Estimating from the most recent retransmission gives underestimates because the packet from a previous transmission sometimes comes back just after a retransmission. Both mistakes feed upon themselves. High RTOs tend to cause higher RTTs. Low RTO estimates tend to cause measured RTTs to get even lower. (The author refers to this problem as "RTO feedback".)

The problem is one of insufficient information. It is hard to figure out which RTT values are misleading. RDP has a small advantage over TCP here because it tracks every packet separately, while TCP implementations usually track the oldest outstanding packet. (The author attempted to exploit RDP's better tracking mechanisms to cull out bad RTTs.)

In the implementation, every RDP packet has an associated timer and retransmission counter. The timer counts up from zero, where zero is the time the packet was first sent. The retransmission counter is set to zero when the first copy of the packet is sent and incremented every time the packet is retransmitted. Packets are retransmitted whenever the timer exceeds the value $(r+1) \times RTO$ where r is the value of the retransmission counter. The RTO is continuously updated with new RTT values, so the retransmission periods will change while the packet is in flight.

Notice that the algorithm used to decide when to retransmit is linear. Many people recommend that this algorithm be exponential, for example, using $(r^{1.5}+1) \times RTO$. Exponential algorithms are felt to deal better with rapidly changing network conditions.¹¹ This implementation uses a linear algorithm for two reasons. First, because the RTO depends on itself, exponential algorithms make it easier for the RTO to grow unreasonably high. Second, the author suspects that because the retransmission period can adjust during the interval between retransmissions, the linear algorithm is sufficiently adaptive.

Another problem with any timing algorithm is choosing an initial estimate for the SRTT and RTO. Overestimating causes excessive delays early in the connection. Underestimating causes excessive retransmissions. The algorithm used was recommended by Zhang. The first packet is sent using exponential timeouts from a fixed table (which can be adapted to the local environment). The round-trip time for this packet is then made the first SRTT and is used to

¹⁰ Among other observations, Mills concluded that the algorithm should be very willing to increase the SRTT when RTTs are increasing, but more cautious about lowering the SRTT.

¹¹ The argument in favor of exponential algorithms is that if a packet has not been acknowledged after one or two transmissions, it is likely that the network path is unstable. In such situations it makes sense to delay introducing more data until the network has had time to settle down.

calculate the initial RTO. This solution is not perfect because the RTT for the first packet includes any one time costs for establishing the link-level connection to the destination, but it works reasonably well in practice.¹²

Another issue was clock granularity. The BSD operating system provides two protocol clocks, one which counts in fifths of a second, the other in half seconds; the implementor must decide which clock to use to maintain the retransmission timers. Clearly, the finer the granularity of the timers, the better the SRTT estimate. But a protocol which updates its timers every fifth of a second is consuming considerably more system resources than one which works in half seconds. Furthermore, the SRTT is a rough upper limit on retransmission time and does not need to be finely tuned. So this implementation uses the half-second timer to track retransmissions. Lossy networks with low round-trip times may suffer disproportionately as a result.¹³

Finally some thought given to trying to weed out misleading RTT values. RDP has more information about the behaviour of its packets than most other protocols. Furthermore, each packet is acknowledged when it is received. As a result, when a packet is acknowledged, in addition to computing the RTT, RDP can also determine if the packet has been retransmitted and whether the packet has been acknowledged in the same order in which it was sent (i.e., whether any packets sent after it have been acknowledged).

Knowing whether a packet has been retransmitted looks like useful information, but in practice it is difficult to apply. The problem is that there is no way to tell which transmission of the packet is being acknowledged. Given an acknowledgement and the knowledge that it is for a packet that has been transmitted n times, for what values of n should the RTT be viewed as misleading? Clearly if $n=1$ the RTT should be kept, and if n is quite large, the RTT should be ignored. But where is the line drawn? Choosing a limit m on n implies choosing a maximum variation in the RTT; if the round-trip time suddenly soars to more than $m \times RTO$, it will be ignored. Any value for m is likely to be wrong in some situation.

Looking at whether the packet has been acknowledged in order is more promising. Networks tend to preserve order, although most networks will occasionally deliver packets out of sequence. So if a packet is acknowledged later than packets which were sent after it, it is likely that the acknowledgement is for a retransmission of the packet. Since RDP tracks retransmissions, this assumption can be checked. This gives a possible RTT weeding algorithm. If s is the sequence number of the packet being acknowledged, h is the highest sequence number for which an acknowledgement has been received, and r is the retransmission counter for packet s , the algorithm is:

If $s < h$ and $r > 0$ discard the RTT

Testing suggests that this algorithm works well.

4.6. A Surprise

The design stage turned up one major surprise. At the start of this project, the author expected that unordered delivery of records would be substantially more efficient than ordered delivery because he assumed that there would be considerable overhead involved in keeping records in order. This assumption was incorrect. The use of ring buffers made ordering an $O(1)$ process; a packet's location in the ring buffer is determined by its sequence number and no additional sorting is required.¹⁴

¹² Other methods for choosing the initial SRTT include maintaining a database of old round-trip times for different destinations, which requires some mechanism for managing and updating the database, and using a fixed initial RTO period, which is easy, but guaranteed to be the wrong value for most connections.

¹³ Of course, no matter how fine the clock, there is a point where a high-speed lossy network will suffer. Fortunately, lossy high-speed networks are not common.

¹⁴ It may be useful to explain the system in more detail. The ring buffer is r packets long, where r is the range size. The first position in the buffer always corresponds to the unreceived packet with the lowest sequence number, s . This means that for any packet received, its position in the buffer is the packet's sequence number, p , less s (as adjusted by the ring buffer pointer). Note that $(p-s) \leq r$ is always true.

This is not to say that there is no difference between sequenced and unsequenced mode. Unsequenced delivery requires no queuing of inbound data at the transport layer, while sequenced delivery may require a considerable amount of queuing (proportional to $r \times m$, where r is the range size and m is the maximum packet size).

5. Preliminary Evaluations

It is too soon to talk in definitive terms about RDP's utility as a transport protocol. Such decisions are really communal; programmers experiment with using the protocol and eventually come to some consensus about its merits vs. other available protocols. Since this implementation is just beginning to be distributed, the network community is nowhere near such a consensus.

However, it is possible to look at some of the testing and evaluation that has been done to date. Much of this work is still in progress, so this discussion is preliminary. The results discussed below come from tests run in early 1987, which were designed to shake out performance problems in the initial implementation using the old 32-bit checksum. A certain amount of tuning of the code has been done since then and the checksum has been changed, but the testing of those changes is still in progress. The new tests are expected to show that RDP performance has improved.

5.1. Comparison With Other Protocols

Comparing transport protocols is always a uncertain activity because there is no good standard for making the comparison. Analyzing the behaviour of implementations of two protocols may only indicate that one implementation is less good than the other. But the comparison of two protocols based only on their specifications is likely to miss key differences which may only be discovered during the implementation stage. The best compromise seems to do a little of both: discuss the expected behavior, and then test those expectations by experimenting with implementations.

There are at least two transport protocols with which RDP should be compared: NETBLT and TCP. NETBLT is an experimental transport protocol developed at MIT and, like RDP, is designed to provide high speed transfer of bulk data [1]. TCP is the most commonly used transport protocol on IP networks. Unlike RDP and NETBLT, TCP is designed to be a generally useful transport protocol, and is not explicitly tuned for bulk data transfer. Nonetheless, TCP is often used for bulk transfer, and thus a comparison is of interest.

NETBLT takes a radically different approach from RDP and TCP to the problem of bulk data transfer. It is designed to transfer large buffers of data, where a buffer can potentially be 2^{32} bytes long. NETBLT breaks a buffer down into conveniently sized packets for transmission. The packets are sent in a series of bursts, where the bursts of packets are carefully spaced to avoid overloading the capacity of the network. Detection of lost packets is the responsibility of the receiver, which must request the retransmissions of missing packets.

NETBLT's approach to the bulk transfer problem is novel, and it is a research project worth watching. The author does not have access to a NETBLT implementation and is thus unable to make any definitive comparisons. Interesting points of comparison would include the different flow control mechanisms and acknowledgement strategies.

TCP has been in heavy use for several years and is easier to analyze. TCP implements a reliable byte stream between two peers. Flow control on the stream is managed through use of a window size, measured in bytes, which may be dynamically changed by the receiver. Each TCP data packet contains a contiguous section of the byte stream; within some limits, TCP implementations are free to use whatever packet size is convenient. Note that the contents of packets may overlap, i.e., a sequence of bytes may appear in more than one packet. As a result, the receiver must be prepared to do some moderately complex splicing of packets to reconstitute the stream. Data is acknowledged by returning the number of the highest numbered byte received in order.

There is no extended acknowledgement mechanism so TCP can retransmit only the packet containing the oldest unacknowledged byte. Furthermore, round-trip times are only kept for the oldest byte.

In general, one would expect TCP to transfer data faster than RDP when the application is sending data in small chunks because TCP can combine the application's data into larger, more efficient packet sizes, while RDP cannot. Because it provides dynamic flow control, TCP also deals better with peers with mismatched speeds.

RDP could be expected to outperform TCP when the record sizes are large or the packet loss rate is significant. Because each RDP packet is a discrete unit that can be acknowledged separately, RDP has the advantage of better timer information, a windowing scheme that adapts better to loss (any packet can be retransmitted, not just the first), and no resequencing costs. Some of the tests below support this analysis.

5.2. Timer Algorithms

Several tests have been run to see if the timer algorithm described in section 4.4 actually keep accurate estimates of the round-trip time. Two questions were of particular interest: (1) does the algorithm keep a good estimate for the RTO, and (2) does the algorithm for selectively discarding RTTs allow the implementation to maintain good RTO values in the face of loss.

To test the accuracy of the RTO estimates, tests were run over connections where the measured round-trip times varied by more than a factor of two, but the loss rate was negligible (around 0.25%). A filter was then put into the connection to randomly drop packets at a fixed rate.

In the first test, the drop rate was set to 0% (all packets got through). One thousand data packets of a fixed size were sent over the connection, and statistics were kept on how many duplicate data packets were received. If more than a handful of packets are sent twice then the RTO is set too low. On average, 0.5% the data packets were retransmitted, which suggests the RTO value was reasonable.

In the second test, the drop rate was set high enough that RTO feedback could occur (dropping 8% of all packets). Again, one thousand data packets were sent and the throughput was measured. The RDP code was conditionally compiled so that it could be set to use every RTT to compute the RTO, or selectively discard RTTs as described in section 4.4. The throughput rates were compared. The selective algorithm gave 62% better throughput. Furthermore, the selective algorithm did not suffer from RTO feedback, while the indiscriminating algorithm did.

5.3. Using Extended Acknowledgements to Reduce Retransmissions

One of the interesting questions about RDP is whether extended acknowledgements are a useful feature of reliable protocols. The timer tests suggest extended acknowledgements are useful for keeping track of round-trip times, but extended acknowledgements were also intended to reduce the number of unnecessary data retransmissions caused by lost acknowledgements. This section discusses a test designed to see how much, if at all, extended acknowledgements reduced retransmissions.

In the test, 100,000 packets were sent over a software loopback that had been rigged to randomly drop a certain percentage of packets. This test was run for loss rates ranging from 0% to 12% with and without extended acknowledgements. The total number of packets transmitted and the number of duplicate data packets received were counted. The results are shown in Figure 2.

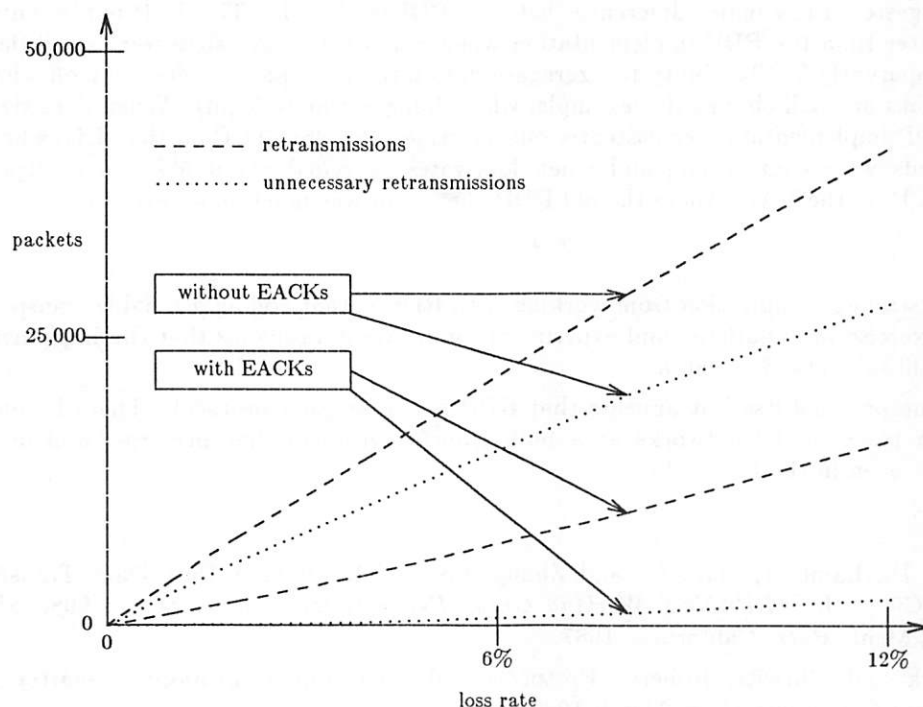


Figure 2: Retransmissions and Extended Acknowledgements

The results are striking. At all loss rates extended acknowledgements reduce the number of retransmissions by roughly 60% and the number of unnecessary retransmissions by 90%.

5.4. Throughput

Of course the most interesting question about RDP is: does it transfer data faster than existing protocols? Right now, because of problems with testing, it isn't possible to give a good answer to this question.

Several tests designed to compare RDP's throughput with TCP's throughput have been run. So far, they have given evidence of performance problems in either the TCP or RDP implementations, which made it difficult to evaluate the test results.

One major problem was that the 4.2 BSD TCP was usually incapable of sustaining a connection when the loss rate exceeded 10%. Even with loss rates in the high single digits, the RTO in the TCP implementation became wildly high. As a result, the RDP implementation (which did not have the timer problems) often gave throughput rates which were twenty times faster than TCP over lossy connections. This is an indication of better timer management, but is not a true test of TCP's merits. The 4.3 BSD implementation apparently does not have these timer problems, but the author only had access to a 4.3 BSD VAX, which had problems with the RDP checksum.

Because of its sensitivity to host bit-ordering the 32-bit RDP checksum routine was slightly faster than the TCP checksum routine on a SUN, but four times slower than the TCP checksum routine on a VAX. This had strange effects on the test results. For example, some tests were run over a fast link with varying loss rates. On the VAX, loss rates of less than 6% had almost no effect on RDP throughput. On the SUN, even a 1% loss rate had an effect on throughput. Since these tests were run, the original checksum algorithm has been replaced with the more predictable 16-bit TCP checksum.

The tests suggested a few major differences between RDP and TCP. The TCP implementation always did better than the RDP implementation when application data sizes were small (less than 64 bytes). Apparently TCP's ability to aggregate data into larger packets does pay off when applications send data in small chunks (for example, when doing a remote login). When data sizes were larger, the RDP implementation consistently outperformed the BSD TCP on the SUN, where the checksum speeds were comparable, and when loss rates exceeded about 5%, even outperformed the BSD TCP on the VAX, where the old RDP checksum was much more expensive.

6. Observations

The author's strongest impression from working with RDP is that coding a reliable transport protocol is still an exercise in hypothesis and experimentation. Many problems that the implementor confronts are still subjects of research.

Looking at the protocol itself, it appears that RDP is a promising protocol. There is some hope that it has a place on IP networks as a bulk transfer protocol that performs well on a variety of networks, even in the face of loss.

Bibliography

1. Clark, David D., Lambert, Mark L., and Zhang, Lixia. NETBLT: A Bulk Data Transfer Protocol; RFC998. In *ARPANET Working Group Requests for Comments*, no. 998. SRI International, Menlo Park, California. 1987.
2. Haverty, Jack and Gurwitz, Robert. Protocols and their implementation: A matter of choice. In *Data Communications*, March 1983.
3. Hinden, Robert M. A Host Monitoring Protocol; RFC869. In *ARPANET Working Group Requests for Comments*, no. 869. SRI International, Menlo Park, California. 1983.
4. Hinden, Robert M. and Partridge, Craig. Version 2 of the Reliable Data Protocol; (draft RFC). 1987.
5. Jain, Raj. A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks In *IEEE Journal On Selected Areas in Communications*, Vol. 4, No. 7. 1986.
6. Leffler, Samuel J., Joy, William N., Fabry, Robert S., and Karels, Michael J. Networking Implementation Notes, 4.3BSD Edition. In *UNIX System Manager's Manual* (4.3 Berkeley Software Distribution). University of California, Berkeley, California. 1986.
7. Leffler, Samuel J., Fabry, Robert S., Joy, William N., Lapsley, Phil, Miller, Steve and Torek, Chris. An Advanced 4.3BSD Interprocess Communication Tutorial. In *UNIX Programmer's Supplementary Documents*, Volume I (4.3 Berkeley Software Distribution) University of California, Berkeley, California. 1986.
8. Mills, David. Internet Delay Experiments; RFC889. In *ARPANET Working Group Requests for Comments*, no. 889. SRI International, Menlo Park, California. 1983.
9. Nagle, John. Congestion Control in IP/TCP Networks; RFC896. In *ARPANET Working Group Requests for Comments*, no. 896. SRI International, Menlo Park, California. 1984.
10. Perry, Dennis G. Congestion in the Arpanet. Letter to the TCP-IP Mailing List. October 1, 1986.
11. Postel, Jon, ed. Internet Protocol; RFC791. In *ARPANET Working Group Requests for Comments*, no. 791. SRI International, Menlo Park, California. 1981.
12. Postel, Jon. Internet Control Message Protocol; RFC792. In *ARPANET Working Group Requests for Comments*, no. 792. SRI International, Menlo Park, California. 1981.
13. Postel, Jon, ed. Transmission Control Protocol; RFC793. In *ARPANET Working Group Requests for Comments*, no. 793. SRI International, Menlo Park, California. 1981.
14. Postel, Jon. Simple Mail Transfer Protocol; RFC821. In *ARPANET Working Group Requests for Comments*, no. 821. SRI International, Menlo Park, California. 1982.

15. Tanenbaum, Andrew S. *Computer Networks*. Prentice Hall, Inc. Englewood Cliffs, New Jersey. 1981. pp. 215-222.
16. Velten, David, Hinden, Robert and Sax, Jack. Reliable Data Protocol; RFC908. In *ARPANET Working Group Requests for Comments*, no. 908. SRI International, Menlo Park, California. 1984.
17. Watson, Richard W. Timer-Based Mechanisms in Reliable Transport Protocol Connection Management. In *Computer Networks*, 1981. North-Holland Publishing Company.
18. Zhang, Lixia. Why TCP Timers Don't Work Well. In *Proceedings of SIGCOMM '86*. Association for Computing Machinery. 1986.

REMOTE UNIX TURNING IDLE WORKSTATIONS INTO CYCLE SERVERS

Michael J. Litzkow

University of Wisconsin
Computer Sciences Department
Project Topaz
ucbvax!uwvax!mike or mike@wisc.edu

ABSTRACT

A computing environment consisting of workstations connected by a local area network is now common. Often these workstations are assigned to individual users, and thus represent a significant unused resource when those individuals are not working. Remote Unix, (RU) uses these idle workstations to execute compute-bound jobs in the background. Users submit jobs to RU from their own workstations. The jobs are queued, and eventually executed remotely on idle workstations. When their jobs have completed, users are notified by mail. The owner of a workstation has absolute priority over RU jobs. When the owner initiates interactive or other non-RU work, any RU job is automatically checkpointed and restarted on another workstation. RU jobs may go through any number of checkpoints before eventual completion. Finding idle workstations, handling of remote system calls, and checkpointing when necessary are all handled by the RU software without intervention from users. In addition to providing "free" cycles, RU allows completion of very long running jobs which might otherwise be aborted by system crashes and shutdowns. The longest running job so far has completed successfully after accumulating 60 CPU days over a 3 month period. This paper describes the computing environment for which RU was designed, and current limitations on the class of jobs which it can handle. The three main components of RU — remote system call handling, a general UNIX[†] checkpointing facility, and distributed spooling and control — are each discussed. The current version of RU supports only single process jobs. Possible extensions are discussed.

1. Introduction

The advent of workstations and local area networks has had a dramatic impact on productivity in the UNIX environment over the past several years. Unfortunately, many of these same workstations which are so productive during the work day, are often completely wasted at other times. Remote Unix is a facility which allows these "wasted" cycles to be used productively by compute-bound jobs.

1.1. Our Local Environment

The University of Wisconsin Computer Sciences department currently has about 100 MicroVaxII^{††} workstations, several VAX 11/750's^{††}, and two VAX 11/780's^{††}. All are connected by a local area network, run the same version of UNIX, and are object-code compatible. The majority of the workstations are placed in individual's offices, and are dedicated to the use of that individual, (the "owner"). Owners expect to have full use of their workstation at any time, day or night. Most usage is done in the owner's office, but occasionally users log in from other machines or dial up from home.

We also have a number of users who are interested in running large, compute bound jobs (batch jobs), such as simulation programs. Often a user will want to run many copies of a long-running simulation with slightly different parameters.

We have developed the Remote Unix (RU) facility to accommodate batch jobs by using the otherwise idle workstations in a way which does not interfere with the normal work of the workstation's owner. The RU package consists of facilities for queuing and execution control, automatic checkpointing, and

[†]UNIX is a trademark of AT&T Bell Laboratories

^{††}MicroVaxII, VAX 11/750, and VAX 11/780 are trademarks of Digital Equipment Corporation

remote execution of batch jobs. Users submit jobs to the RU queue, and when some workstation becomes "idle", the job is remotely executed on the idle workstation. Most system calls are done remotely, so that file accesses and user ID's refer to the environment in which the job was submitted. When the remote workstation's owner wants to regain use of the workstation, the job is automatically checkpointed to a file on the originating machine. When some other workstation becomes idle, execution can resume. The system takes care of re-opening all the files in the correct modes, assigning the original file descriptor numbers to them, and seeking to the appropriate places. When the job completes, the user is sent mail regarding its completion status. Some implementation details and current limitations are discussed below. RU is implemented entirely through special libraries and user level programs; it does not depend on any modifications of the UNIX kernel.

2. Remote Execution

RU jobs are linked with a special version of the C runtime library, in which most of the system call "stubs" have been replaced with remote execution code. This code marshals the arguments, sends the request over the net to a "shadow" process on the originating machine, and places the results into the correct user area(s). A few system calls (e.g. `sbrk`) are done locally.

When a job is to be executed remotely, a shadow process is created on the originating machine. The shadow executes a simple program called the "starter" on the remote machine using the standard C library routine `rcmd(3)`. The starter copies the core image to a special directory on the remote machine, sets up pre-defined file descriptors as network connections to the shadow, starts execution of the job, then waits for it to exit. When the job completes, the core image on the remote machine is removed by the starter. The core image is in a format which is suitable for `exec`, but contains extra information regarding currently open files, contents of registers, and current working directory. The RU library contains startup code which opens the necessary files corresponding to `stdin`, `stdout`, and `stderr` before "main" is called.

2.1. Remote execution limitations

The current system does not support the UNIX system calls `pipe()`, `fork()`, `exec()`, signals, or inter-process communication. Programs which depend on knowing their own process id will not work correctly because the id of an RU process changes after each checkpoint. Programs which use "wall clock time" will not produce re-producible results. Programs which do large amounts of I/O are not appropriate for RU, since the overhead of the remote read and write operations is prohibitive.

Despite all the limitations, there are a large number of compute intensive jobs which are well suited to RU. We have also been able to run some jobs which could not practically be run on a normal UNIX system because of the large amount of CPU time required.

3. Checkpointing

When a workstation owner resumes use of that workstation while it is executing an RU job, the job is stopped and checkpointed on the originating machine. This allows preemption of jobs without loss of work, and provides fault-tolerance. Should the system crash before an RU job is finished, the job will automatically be restarted from the latest checkpoint.

Checkpointing is accomplished via the remote system call mechanism in response to a signal. A signal handler is supplied by the RU library which catches the signal, and writes out the necessary information. The registers are saved on the stack, and the open files and other special data are stored in the program's data segment. The program's data and stack are then written to the checkpoint file via the remote write system call. The checkpoint file is written in a UNIX executable format, and the entry point is set to a routine called "restart".

A program is restarted from a checkpoint file by the "exec" system call, just like any other UNIX executable. The restart routine changes to the current working directory, re-opens the files and seeks to the appropriate places, restores the stack and registers from the checkpoint file, and continues execution from where it was suspended at the time of the checkpoint.

4. Spooling and Control

Each machine maintains its own queue of RU jobs which it wishes to run remotely. Users submit jobs via the "ru" program, giving program arguments, and possibly specifying files to be used in place of stdin, stdout, and stderr. If the user does not specify redirections for the standard files, default names are generated (e.g. /dev/null for stdin). Programs are also provided for displaying the RU queue, changing priority of programs within the queue, and removing RU jobs prior to completion.

The control system consists of a "central resource manager", which gathers general information about all the machines, and a "local scheduler" which makes decisions affecting only a particular workstation.

The resource manager periodically polls all the schedulers, determining which workstations are accepting ru jobs, which have jobs to run, and how many individual users are waiting for those jobs. Since the resource manager requires only summary information from the other machines, polling can proceed quickly and without much communication overhead. Upon finding an "idle" workstation the manager chooses another workstation which wants to run a job, and sends it permission to run a job on the idle machine. The machine which receives permission to run an RU job remotely has responsibility for choosing among the jobs it wishes to run, and communicating directly with the idle workstation to initiate execution.

Each workstation in the RU "pool" runs a program called the local scheduler. The local scheduler is responsible for determining whether or not the workstation on which it is running is "idle", and for getting rid of RU jobs when the workstation's owner returns to work. This is accomplished by a finite state machine within the scheduler. Periodically the scheduler wakes up, examines the process queue on its machine, evaluates the load, and updates the state if needed. Only jobs belonging to ordinary users, (not system processes or RU jobs), are included in this load evaluation. When the load has been low for a period of time, the scheduler determines that its machine is idle and can accept ru jobs. At this point the scheduler's state machine enters the "accept" state. When the machine acquires an RU job to run from another machine, its state becomes "RU". Only one RU job is accepted at any one time. If a user returns to the workstation and attempts to use it, any RU job which is running will be temporarily stopped. Assuming the user continues to use the workstation, the RU job will later be checkpointed back to its originating machine, and the workstation will return to the "user" state. If the user only uses the workstation for a short time, then leaves again, the RU job will be resumed, and the workstation will return to the "RU" state. The scheduler also accepts permission to run RU jobs remotely, and chooses which job to run when given permission.

A side benefit of the scheduler is a "screen saver". Whenever the workstation is "idle", i.e. ready to accept RU work or doing RU work, the screen saver is put on the workstation's monitor. The screen saver also notices any keyboard activity, and informs the scheduler that the machine is no longer idle. Thus workstation owners can just walk away from their machines at any time, leaving them available for RU, and can regain control immediately when they return.

Due to the partially distributed control system, a two level priority system is used. The resource manager implements a priority scheme based on how many separate users have jobs queued on each workstation, and how long those jobs have been waiting to run. When there are idle machines, the resource manager chooses the machine which has had the most users waiting the longest time, and gives it permission to remotely execute a job on the idle machine.

The scheduler which receives such a permission must choose a job from its local queue to be run. Each process in the queue has a priority. Users can specify what priority their jobs should have when they are submitted, but only the super user can specify a priority greater than the default. If there are more than one job at the highest priority level, the scheduler will choose the oldest one.

5. Future Work

The primary limitation of the current RU is the range of system calls supported. Eventually, we plan to implement all UNIX system calls except possibly the networking calls. We have already begun work on signal implementation, which is not too difficult. The main steps are discovering all the caught, blocked, and ignored signals, as well as any pending signals at the time of checkpointing so that they can be restored during the restart procedure. Fork can most easily be implemented by executing a fork in the shadow as well as the remote job. Thereafter each job is separate, has its own checkpoint file, etc. The simplest

implementation would be to require that all processes in a "family" be loaded on a particular remote machine at the same time. Exec could be implemented by overwriting an RU job's checkpoint file with a new one, then using the UNIX exec system call to overwrite the executing core image with the new one. Pipe will require special handling at checkpoint time to "drain" the pipe and save those contents somewhere in the checkpoint file.

Another useful enhancement would be access to "local" files for certain applications. For example, font-description files are replicated on all machines, so a typesetting program should access the local copy rather than the copy on the originating machine.

Acknowledgements

This project is based on the idea of a "processor bank", which was introduced by Maurice Wilkes in connection with his work on the Cambridge Ring.¹

I would like to thank Don Nuehengen and Tom Virgilio for their pioneering work on the remote system call implementation; Matt Mutka for first convincing me that a generalized checkpointing method could be practical and for ideas on how to distribute the control and prioritize the jobs; Marvin Solomon whose "red pen" contributed greatly to the readability of this paper; and the countless others who have contributed their help and ideas to the project.

This research was supported by the National Science Foundation under grants MCS81-05904 and DCR-8512862 and by a Digital Equipment Corporation External Research Grant.

Wilkes, M. V., Invited Keynote Address, 10th Annual International Symposium on Computer Architecture, June 1983.

The Network Computing Architecture and System: An Environment for Developing Distributed Applications

Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin,
Joseph N. Pato, Geoffrey L. Wyant
Apollo Computer Inc.

...!{wangins,yale,mit-eddie}!apollo!mishkin

1. Introduction

The Network Computing Architecture (NCA) is an object-oriented framework for developing distributed applications. The Network Computing System[™] (NCS[™]) is a portable implementation of that architecture that runs on Unix[®] and other systems. By adopting an object-oriented approach, we encourage application designers to think in terms of what they want their applications to operate on, not what server they want the applications to make calls to or how those calls are implemented. This design increases robustness and flexibility in a changing environment.

NCS currently runs under Apollo's DOMAIN/IX[™] [Leach 83], 4.2BSD and 4.3BSD, and Sun's version of Unix. Implementations are currently in progress for the IBM PC[®] and VAX/VMS[®]. Apollo Computer has placed NCA in the public domain.

In addition to its object orientation, some interesting features of the system are as follows. It supplies a transport-independent remote procedure call (RPC) facility using BSD sockets as the interface to any datagram facility. It provides at-most-once semantics over the datagram layer, with optimizations if an operation is declared to be idempotent. It is built on top of a concurrent programming support package that provides multiple threads of execution in a single address space, although versions can be made for machines that just have asynchronous timer interrupts. The data representation supports multiple scalar data formats, so that similar machines do not have to convert data to a canonical form, but can instead use their common data formats. The RPC interface definition compiler is extensible. Procedures to do the client/server binding can be attached to data types defined in the interface. Also, complex data types can be marshalled by user-supplied procedures which convert such types to data types the compiler understands. There is a replicated global location database: Using it, the locations of an object can be determined given its object ID, its type, or one of its supported interfaces.

There are several motivations for NCA. Large, heterogeneous networks are becoming more common. Users of systems in such networks are often frustrated by the fact that they can't get those systems to work cooperatively. Over the last few years, advances have been made in allowing *data sharing* to occur between the systems, but not *compute sharing*. Tools to allow the effective use of the aggregate compute power have not been available. The inability to share computing resources has become even more aggravating as more specialized processors (e.g. ones designed to run numerical applications fast) have become more widespread. Current "technology" obliges users of those processors to resort to FTP and Telnet. Even in an environment of systems of relatively similar power, a network computing architecture is called for: There are applications that can take advantage of many systems in parallel. (Parallel "make" is the most obvious example.) Also, replicating resources over a number of machines increases the reliability seen by users of the network.

It is important to understand that there is almost no "network application" that can't be implemented *without* NCA/NCS. However, the implementation is bound to be more difficult, less general, and harder to install on a variety of systems. Further, experience has shown that some obviously useful network applications simply don't get written because of these problems. The existence of NCA/NCS helps to solve these problems and as a result, expand the set of network applications.

2. Architecture

Figure (1) illustrates NCA's overall structure.

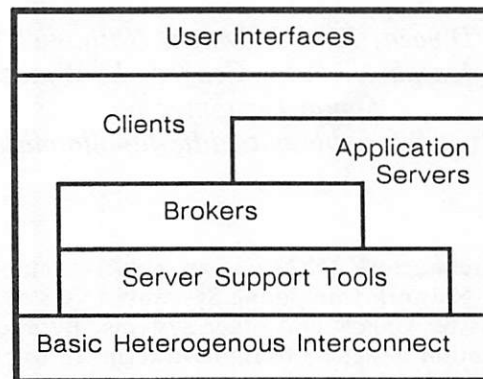


Figure 1. NCA's overall structure.

2.1 Heterogeneous Interconnect

The lowest level provides the basic interconnection to heterogenous computing systems. At this layer NCA currently defines a remote procedure call protocol (NCA/RPC), a Network Interface Definition Language (NIDL), and a Network Data Representation (NDR). RPC is a mechanism that allows programs to make calls to subroutines where the caller and the subroutine run in different processes, most commonly on different machines. The RPC approach and an implementation similar to ours is described in detail by Birrell and Nelson [Birrell 84]. NIDL is a high-level language used to specify the interfaces to procedures that are to be invoked through the RPC mechanism. NCS includes a portable NIDL compiler that takes NIDL interfaces as input and produces stub procedures that, among other things, handle data representation issues and connect program calls to the NCS RPC runtime environment that implements the NCA/RPC protocol. The relationships among the client (i.e. the caller of a remotized procedure), server, stubs, and NCS runtime is shown in figure (2).

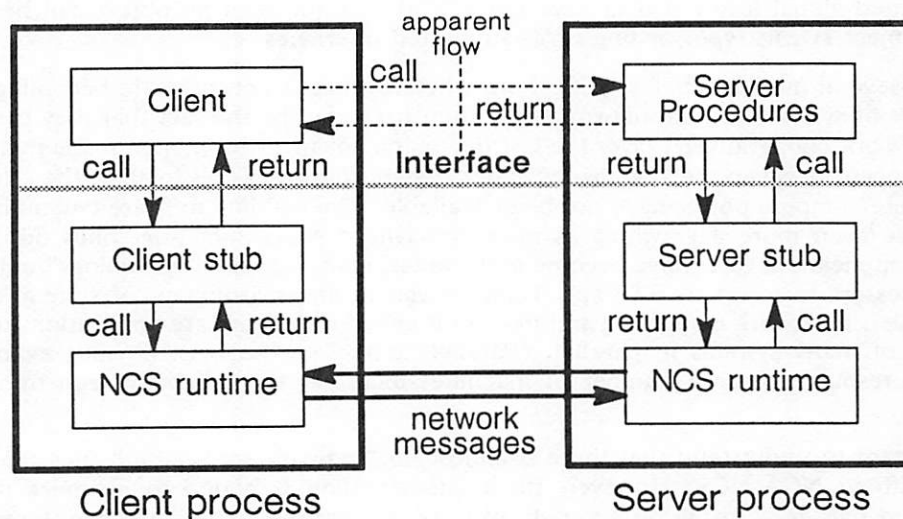


Figure 2. Relationships among client, server, stubs and NCS runtime

2.2 Server Support Tools

Augmenting the heterogenous interconnect layer are the server support tools. These tools simplify the writing of complex applications in a distributed environment. Currently these consist of the

Data Replication Manager (DRM) and Concurrent Programming Support (CPS). DRM provides a weakly consistent, replicated database facility. It is useful for providing replicated objects when high availability is important and weak consistency can be tolerated. CPS provides integrated lightweight tasking facilities. CPS allows multi-threaded servers to be written easily.

2.3 Brokers, Clients, Servers and User Interfaces

Built on top of the server-support tools are a set of *brokers*. A broker is a third party agent that facilitates transactions between principals. In a network computing environment brokers are primarily useful in determining object locations, but can also be used for establishing secure communications (i.e. authentication), associatively selecting objects, issuing software licenses, and a variety of other administrative chores not directly related to the operation of the principals. The role of brokers is shown in figure (3).

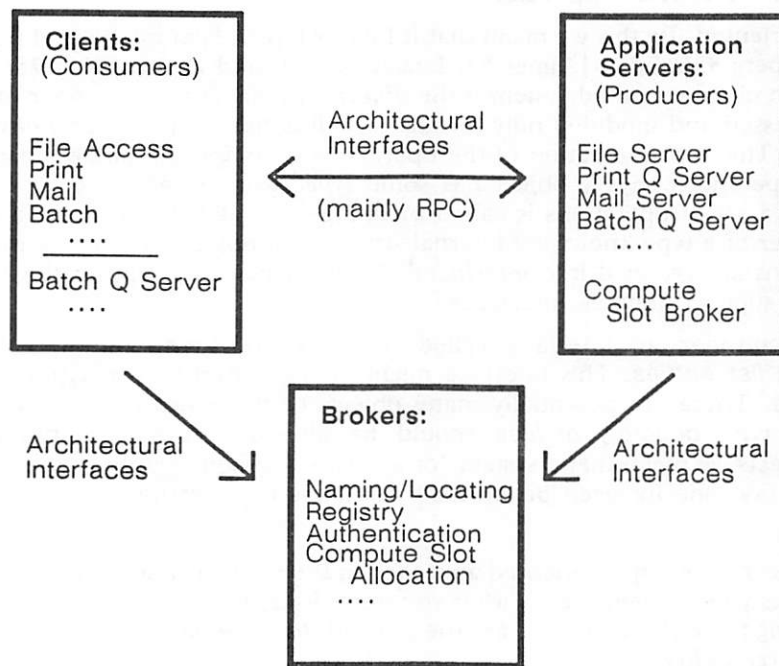


Figure 3. The role of brokers in NCA

Client programs and application servers make use of the three base layers. Application servers are the "producers" of services and clients the "consumers". Servers invoke brokers to make their existence known. Clients can invoke brokers to locate application servers and then use the underlying RPC mechanism to make use of the services provided. The application server may be in turn a client of other distributed services.

From user's perspective, user interfaces tie all the pieces together. However, user interfaces are not part of NCA and will not be discussed in this paper.

2.4 Unique Identifiers

An important aspect of NCA is its use of *universal unique identifiers* (UUIDs) as the most primitive means of identifying NCA entities (e.g. objects, interfaces, operations). UUIDs are an extension of the unique identifiers (UIDs) already used throughout Apollo's system [Leach 82]. Both UIDs and UUIDs are fixed length identifiers that are guaranteed to refer to just one thing for all time. The principal advantages of using any kind of unique identifiers over using string names at the lowest level of the system include: small size, ease of embedding in data structures, location transparency, and the ability to layer various naming strategies on top of the primitive naming mechanism. Also, identifiers can be generated anywhere, without first having to contact some

other agent (e.g. a special server on the network, or a human representative of a company that hands out identifiers).

UIDs are 64 bits long and are guaranteed to be unique across all Apollo systems by embedding in them the node number of the system that generated the UID and the time on that system that the UID was generated. To make it possible to generate unique identifiers on non-Apollo system we defined UUIDs to be 128 bits and made the encoding of the identity of the system that generates the UUID more flexible.

The remainder of this paper discusses several aspects of NCA and NCS: NCA's object-oriented approach; NIDL; NDR; the NIDL compiler; the Location Broker used in connecting clients with servers; and the networking model and protocol used by NCS. We conclude with a description of future directions we expect NCA and NCS to follow.

3. The Object-Oriented Approach

NCA is object-oriented. By this we mean that it follows a paradigm established by systems such as Smalltalk [Goldberg 83], Eden [Almes 83, Lazowska 81], and Hydra [Wulf 75, Cohen 75]. The basic entity in an object-oriented system is the *object*. An object is a container of state (i.e. data) that can be accessed and modified only through a well-defined set of *operations* (what Smalltalk calls *messages*). The implementation of the operations is completely hidden from the client (i.e. caller) of the operations. Every object has some *type* (what Smalltalk calls a *class*). The implementation of a set of operations is called a *manager* (what Smalltalk calls a set of *methods*). Only the manager of a type knows the internal structure of objects of the type it manages. Sets of related operations are grouped into *interfaces*. Several types may support the same interface; a single type may support multiple interfaces.

For example, consider an interface called *directory* containing the operations *add_entry*, *drop_entry*, and *list_entries*. This interface might be supported by two types: *directory_of_files* and *print_queue*. There are potentially many objects of these two types. That there are many objects of the type *directory_of_files* should be obvious. By saying that there are many *print_queue* objects we mean that a system (or a network of connected systems) might have many print queues — say, one for each department in a large organization.

3.1 Motivation

The reason for using the object-oriented approach in the context of a network architecture is that this approach lets you concentrate on *what* you want done, instead of *where* it's going to be done and *how* it's going to be done: objects are the units of *distribution*, *abstraction*, *extension*, *reconfiguration*, and *reliability*.

Distribution. Distribution addresses the question of where an operation is performed. The answer to this question is that the operation is performed where the object resides. For example, if the print queue lives on system A, then an attempt to add an entry to the queue from system B must be implemented by making a remote procedure call from system B to system A. (This implementation fact is hidden from the program attempting to add the entry.)

Abstraction. Abstraction addresses the question of how an operation is performed. In NCA, the object's type manager knows how the operation is performed. For example, a single program *list_directory* could be used to list both the contents of a file system directory and the contents of a print queue. The program simply calls the *list_entries* operation. The type managers for the two types of objects might represent their information in completely different ways (because, say, of the different performance characteristics required). However, the *list_directory* program uses only the abstract operation and is insulated from the details of a particular type's implementation.

Extension. The object-oriented approach allows extension; i.e. it specifies how the system is enhanced. In NCA, there are two kinds of extensions allowed. The first is extension by creation of new types. For example, users can create new types of objects that support the *directory* interface; programs like *list_directory* that are clients of this interface simply work on objects of the new type, without modification. The second kind of extension is extension by creation of new interfaces. A new interface is the expression of new functionality.

Reconfiguration. Because of partial failures, or for load balancing, networked systems sometimes need to be reconfigured. In object-oriented terms, this reconfiguration takes place by moving

objects to new locations. For example, if the system that was the home for some print queue failed because of a hardware problem, the system would be reconfigured by moving the print queue object to a new system (and informing the network of the object's new location).

Reliability. The availability of many systems in a network should result in increased reliability. NCA's approach is to foster increased reliability by allowing objects to be replicated. Replication increases the probability that least one copy of the object will be available to users of the object. To make replication feasible, NCS provides tools to keep multiple replicas of an object in sync.

While NCA is object-oriented and we believe that applications that use the object-oriented capabilities of NCA will be more robust and general than those that don't, it is easy to use NCS as a conventional RPC system, ignoring its object-oriented features.

4. Network Interface Definition Language

The Network Interface Definition Language (NIDL) is the language used in the Network Computing Architecture to describe the remote interfaces called by clients and provided by servers. Interfaces described in NIDL are checked and translated by the NIDL compiler.

NIDL is strictly a *declarative* language — it has no executable constructs. NIDL contains only constructs for defining the constants, types, and operations of an interface. NIDL is more than an interface definition language however. It is also a *network* interface definition language and, therefore, it enforces the restrictions inherent in a distributed computing model (e.g. lack of shared memory).

4.1 NIDL Language Constructs

A NIDL interface contains an header, constant and type definitions, and operation descriptions. The header provides the interface identification: its UUID, name, and version number. The UUID is the "name" by which an interface is known within NCA. It is similar to the program number in other RPC systems, except that it is not centrally assigned. The interface name is a string name for the interface which is used by the NIDL compiler in naming certain publicly known variables. The version number is used to support compatible enhancements of interfaces.

A standard set of programming language types is provided. Integers (signed and unsigned) come in one, two, four, and eight byte sizes. Single (four byte) and double (eight byte) precision floating point numbers are available. Other scalars include signed and unsigned characters, as well as booleans and enumerations.

In addition to scalar types, NIDL provides the usual type constructors: structures, unions, pointers, and arrays. Unions must be discriminated. (I.e. non-discriminated unions are not permitted. The actual data values must be known at runtime so that it can be correctly transmitted to the remote server.) Pointers, in general, are restricted to being "top-level." That is, pointers to other pointers, or records containing pointers are not permitted. Later, we'll see how this restriction can be relaxed. Arrays can be fixed in size or have their size determined at runtime.

Operation declarations are the heart of a remote interface definition. These define the procedures and functions that servers implement and to which clients make calls. All operations are strongly typed. This enables the NIDL compiler to generate the code to correctly copy parameters to and from the packet and to do any needed data conversions. Operation declarations can be optionally marked to have certain semantic properties, for example whether they are *idempotent*. (An idempotent procedure is one that can be executed many times with no ill-effect.)

All operations are required to have a *handle* as their first parameter. This parameter is similar to the implicit "self" argument of Smalltalk-80 or the "this" argument of C++ [Stroustrup 86]. The handle argument is used to determine what object and server is to receive the remote call. NIDL defines a primitive handle type named *handle_t*. An argument of this type can be used as an operation's handle parameter. Clients can obtain a *handle_t* by calling the NCS runtime, providing an object UUID and network location as input arguments. Use of more abstract kinds of handles is described below.

Handle arguments can be implicit. An interface definition can declare that a single global variable should be treated as the handle argument for all operations in the interface. While this style

conflicts with some of the goals of the object-oriented approach (e.g. it makes it harder to make calls on different objects using the same interface), it can be useful in cases where an existing local interface is being converted to work remotely.

4.2 NIDL Example

Figure (4) is a short example of an interface described in NIDL. The example is of an interface to a bank object that supports a single operation: deposit money into an account.

(1) Defines the UUID by which this interface is known. This is the first version of this interface. If in the future, new operations are added, the version number should be incremented. (2) Declares the interfaces upon which this interface is dependent. The *import* statement is similar to *#include*, except that the named interface is not textually included. The contents are made available for the importer to refer to types and constants defined in that interface. This allows factoring out a common set of types into a base interface. (3) Defines a set of types (account and account name types) that are used by the bank operations. Finally (4) defines the operation itself.

A variant of NIDL that looks Pascal-like (as opposed to the C-like version of which figure (4) is an example) is also available. Regardless of the variant used as input to the NIDL compiler, the output is the same.

```
[uuid(334033030000.0d.000.00.87.84.00.00.00), version(1)]    (1)
interface bank {
  import
    "nbase.imp.idl";                                          (2)
  typedef                                                     (3)
    long int bank$acct_t;
  typedef
    char bank$acct_name_t[32];
  void bank$deposit(                                          (4)
    [in]   handle_t      h,
    [in]   bank$acct_t   acct,
    [in]   long int      amount,
    [out]  status_t      *status
  );
};
```

Figure 4. Example interface

4.3 Object-Oriented Binding

One drawback of the language as described so far is that all operations are required to have a primitive *handle_t* as their first argument. This means clients need to embed these handles in their programs, and to manage the binding to servers themselves. We would like to achieve as much local-remote transparency as possible (i.e. to make programs insensitive to the location of the objects upon which they operate). Embedding primitive handles in client programs destroys much of this transparency. To relieve clients of the need to manage these handles, we introduced the notion of *object-oriented binding*.

Object-oriented binding comes into play when the first parameter to an operation is *not* a *handle_t*. In this case, the type is taken to represent some more abstract, client-oriented handle. Since to actually make remote calls, a *handle_t* is required, some way is needed to translate the abstract handle into a *handle_t*. The person who creates the abstract type is thus obliged to write a procedure to do the conversion. This procedure is assumed to have the name *type_bind* (where *type* is the type name of the abstract handle) and is automatically called from stubs when the remote call is made. You can view the abstract handle as an *object* (in the Smalltalk sense) which supports the *bind* operation.

To make this more concrete, we could reformulate the above bank example in terms of object-oriented binding. Instead of taking a *handle_t* as its first parameter, *bank\$deposit* could take a bank name, of type *bank\$name*. The NIDL compiler would generate a call to *bank\$name_bind* to translate from a bank name to the primitive *handle_t*. This routine would probably call upon some

sort of naming server to look up the bank location. The bind routine might also choose to cache location information to make later translations faster.

Object-oriented binding hides the details of handle binding from the client and allows interfaces to be designed in a more abstract, client-oriented fashion. This provides a higher level of local-remote transparency than other systems which always require the client to manage handles or explicitly name the remote host on each call.

4.4 Marshalling Complex Types

In the section on NIDL language constructs, we stated that pointers could not be nested. The reason is that such nesting would require the NIDL compiler to generate code to transmit general graph structures. However, permitting only top-level, non-nested pointers can be a severe limitation in the design of an interface. For example, it excludes passing tree data structures to remote procedures.

To provide an escape from this restriction, NIDL allows a type to have an associated "transmissible" type. The transmissible type is a type that the NIDL compiler *does* know how to marshall. Any type that has an associated transmissible type must have a set of procedures to convert that type to and from its transmissible type. In the example of the binary tree, the transmissible type could be an array. The *tree\$to_xmit_rep* procedure would walk the tree to build a representation of it in the array, and the *tree\$from_xmit_rep* procedure would reconstruct the binary tree from the array.

Transmissible types may be associated with any type, not just types using nested pointers. Bitmaps are an example. It may be represented internally as a fixed size array of integers. Even though the NIDL compiler is capable of marshalling this, it may be more efficient to have it transmitted in a run-length encoded (RLE) form. So the bitmap type could have an associated *RLEBitmap* type, and a set of procedures for converting to and from the RLE form.

5. Network Data Representation

Communicating typed values in a heterogenous environment requires a data representation protocol. A data representation protocol defines a mapping between typed values and *byte streams*. A byte stream is a sequence of bytes indexed by nonnegative integers. Examples of data representation protocols are Courier [Xerox 81] and XDR [Sun 86]. A data representation protocol is needed because different machines represent data differently. For example, VAXes represent integers with the *least* significant byte at the low address and 68000s represent integers with the *most* significant byte at the low address. A data representation protocol defines the way data is represented so that machines with different local data representation can communicate typed values to each other.

NCA includes a data representation protocol called Network Data Representation (NDR). NDR defines a set of data types and type constructors which can be used to specify ordered sets of typed values. NDR also defines a mapping between ordered sets of values and their representations in messages.

Under NDR, the representation of a set of values consists of two items: a *format label* and a byte stream. The format label defines how scalar values are represented (e.g. VAX or IEEE floating point) in the byte stream; its representation is fixed by NDR as a data structure representable in four bytes.

NDR supports the scalar types *boolean*, *character*, *signed integer*, *unsigned integer*, and *floating point*. Booleans are represented in the byte stream with one byte; *false* is represented by a zero byte and *true* by a non-zero byte. Characters are represented in the byte stream with one byte; either ASCII or EBCDIC codes can be used. Four sizes of signed and unsigned integers are defined: *small*, *short*, *long*, and *hyper*. Small types are represented in the byte stream with one byte, short types with two bytes, long types with four bytes, and hyper types with eight bytes. Either big- or little-endian representation can be used for integers; two's complement is assumed for signed integers. The two sizes of floating point type are *single* and *double*. Single floating point

types are represented with four bytes and double floating point types use eight bytes. The supported floating point representations are IEEE, VAX, Cray, and IBM.

In addition to scalar types, NDR has a set of type constructors for defining aggregate types. These include *fixed size arrays*, *open arrays*, *zero terminated strings*, *records*, and *variant records*.

Fixed sized arrays have a known number of elements. Their values are represented in the byte stream simply as a sequence of representations of the values of the elements. Each element value is represented according to the element type of the array. Open array types have a fixed first index value and element type but their final index value is not known from their type. Therefore, it is necessary to represent the value of the index of the last element in the array immediately before the representation of the values of the array elements.

Zero terminated strings can be viewed as a special case of open arrays; they are open arrays of characters whose last index value is defined by a terminating zero byte. To support this common data type in an efficient manner, NDR represents such values with an explicit length value followed by the characters of the string including the terminating zero character.

Record values are represented in the byte stream by representations of the values of their fields in the order defined by the record type. Variant records are assumed to have an initial set of fixed fields which includes a tag field used to discriminate among the possible variants. Representations of the values of the fields of the selected variant follow the representations of the values of the fixed fields of a variant record value.

Some types may appear to be missing from NDR. NDR has no enumerated types, bit set types, or a pointer type constructor. The definition of a NIDL maps such types onto their representations in an NDR byte stream. For example, NIDL maps enumerated types and bit sets onto the NDR unsigned integer type of the appropriate size. Typed pointer values are mapped into the NDR type which represents the type that the pointer references.

NDR is abstract in that it does not define how the format label and the byte stream are represented in packets. The NIDL compiler and the NCA/RPC protocol are users of NDR: They work together to generate the format label and byte stream, encode the format label in packet headers, fragment the byte stream into packet-sized pieces, and put the fragments in packet bodies.

The important features of NDR are its flexible representation of scalar values, its use of *natural alignment*, and its extensibility.

By using a format label to specify an interpretation of the scalars in a byte stream NDR supports a "recipient makes it right" approach to data conversion in a heterogeneous environment. A sending process can use its preferred encoding of scalars when constructing a byte stream providing that it is one of the defined options. A receiving process needs to convert data representations only when the format specified in the incoming format label differs from its own preferred format. Thus, two compatible machines can communicate efficiently without needing to convert to a conventional network format and back again on each transmission. NDR defines a broadly useful but not universal set of scalar formats. We believe that our choices are reasonable for promoting heterogeneous network computing combining workstations and special purpose server machines. On the other hand, it is important to keep the space of possible formats to a reasonable size because each recipient needs to convert any incoming scalar format to its own.

NDR requires that values be naturally aligned in the byte stream. Natural alignment means that all values of size 2^n are aligned at a byte stream index which is a multiple of 2^n , up to some limiting value of n ; NDR chooses this limit to be 3. (I.e. scalars of size up to eight bytes are naturally aligned.) This permits, but does not require, implementations of NCA to align buffers for the byte stream so that stub code can use natural operators to manipulate values in the byte stream effi-

ciently and without alignment faults. This also helps to promote communication ease between different kinds of machines in a heterogenous environment.

By its use of a format label NDR is an extensible data representation protocol. The format label could be extended to specify other aspects of the data representation such as packing disciplines, dynamic typing schemes, new encodings of scalars, or new classes of scalars.

6. The NCS NIDL Compiler and Stub Functions

NCS includes a compiler which mediates between NIDL on the one hand and NDR and the NCS runtime on the other. The functions of the compiler are: checking the syntax and “semantics” of interface definitions written in NIDL; translating NIDL definitions into declarations in implementation languages such as C; and generating client and server stubs for executing the remote operations of an interface.

The NIDL compiler is organized as a front-end component and a back-end component. The front-end parses and checks an interface definition and produces an abstract syntax tree (AST) intermediate form. If the interface definition is sound, the front-end then passes this tree to the back-end which generates implementation language include files and stub code files for the interface.

NCS's NIDL compiler is implemented for portability in C using YACC and LEX. It is available in source form to encourage its use and extension in heterogeneous networked environments.

6.1 NIDL Compiler Functions

Distributed object-oriented programming imposes certain restrictions on the semantics of interfaces. It is part of the compiler's job (along with the design of NIDL) to enforce these restrictions. We illustrate the front-end's semantic checks with some examples. All types used in a definition must be well defined. All parameters and fields whose type is an open array require the use of a */last_is* attribute to give their size at call time. Every remote interface requires a UUID. Every operation of an interface requires an implicit or explicit handle parameter to support object-oriented programming.

The second major function of the NIDL compiler is to derive files which declare the interface's constants, types, and operations in the languages in which client applications and servers are written. These files are included in client and server programs which use or implement the remote operations of an interface. For the current implementation the supported languages are C and Pascal. Generating these files is done by a fairly straightforward walk over the AST; adding the capability to generate include files in other Algol-like languages would be a simple exercise.

In addition to declaring the constants, types, and operations of an interface, the derived include files declare two important statically initialized variables defined for each interface. One is the *interface specification (ifspec)* which encapsulates the identity of the interface and its salient properties (number of operations, well known ports used, etc.). The ifspec variable is used in the binding and registering operations of the NCS runtime. The second variable is the server *Entry Point Vector (EPV)* which holds pointers to the server side's stub routines. This EPV variable is used by a server process when registering as a server for an interface; it is used by the NCS runtime to dispatch incoming calls.

The third major function of the NIDL compiler is to generate files of stub code for the operations defined in an interface. There are two such files — one contains client side stub routines and the other contains server side stub routines. This emitted code is in standard C, which we use as a universal assembler to promote portability. Each operation in an interface gives rise to a client stub routine and a server stub routine. The following section discusses the functions of these routines.

6.2 Stub Functions

Client stub routines are called by clients of an interface; they have the same interface as the operation for which they stand in. Server stub routines are called by the server side NCS runtime; their interface is defined by NCS. Client stub routines call the client side NCS runtime to perform

remote calls. Server stubs call the manager's implementation of an operation to provide the actual service. Thus, the first function of stubs is to hide the NCS runtime from users and implementors of remote interfaces and to create the illusion of accessing a remote procedure as though it were local.

To communicate input and output arguments and function results between callers and called routines the stub must *marshall* and *unmarshall* argument values into call and reply packets. This is done in accordance with NDR and the conventions of NCS. Unmarshalling code is also responsible for detecting and performing necessary data conversions by comparing the incoming format label with the local formats. Data conversion is done by a combination of inline code and support operations in the NCS runtime.

The stubs also need to calculate the size requirements for call and reply packets based on the dynamic size of input and output arguments. The size information is used to determine whether or not a pre-declared packet on the stack is large enough. If not, the stubs need to allocate and free storage for packets. It is *not* the job of the stub to break up a large packet into pieces that can be sent over the network — the NCS runtime provides the capability of handling arbitrarily sized packets.

Client side stubs map the operations of an interface to the operation number used by the NCS runtime to identify operations; they also pass options designating the desired calling semantics and the ifspec derived from the NIDL declaration of an operation to the NCS runtime's remote call primitive.

On the server side, the stub routines are responsible for managing storage to be used as the server side surrogates for dynamically sized arguments. This is necessary to support the server's illusion of large data structures passed to it by reference.

The stubs also manage the more elaborate features of NIDL described in section 3 above. Client stubs support automatic binding by calling users' binding and unbinding routines when necessary. Implicit handles are made explicit to the NCS runtime by client stub routines. Users' marshalling routines are invoked as necessary by both client and server stubs as part of marshalling input and output arguments of the appropriate types.

In summary, the stub generation function of the NIDL compiler automates the production of a large amount of protocol code based on a routine's interface definition. This is important because the code is complex enough to make its hand coding very error prone and tedious. Hand producing this kind of code has been a major impediment to building distributed systems in the past.

7. Location Broker

A highly available location service is a fundamental component of a distributed system architecture. Objects representing people, resources, or services are transient and mobile in a network environment. Consumers of these entities cannot rely on a priori knowledge of their existence or location, but must consult a dynamic registry. When consumers rely solely on a location service for accessing objects, it becomes essential that the location server remain available in the face of partial network failures.

The *NCA Location Broker* (NCA/LB) protocol is designed to provide a reliable network-wide location broker. This protocol is defined by a NIDL interface and is thereby easily used by any NCA/RPC based application.

The NCA/LB, unlike location services like Xerox SDD's Clearinghouse [Oppen 83] or Berkeley's Internet Name Domain service (BIND) [Terry 84], yields location information based on UUIDs rather than on human readable string names. The advantages of using UUIDs were described earlier.

7.1 Locating

An object's type manager must first advertise its location with the Location Broker in order for that object to be locatable. A manager advertises itself by registering its location and its willingness to support some combination of specific objects, types of objects, or interfaces. A manager can

choose to advertise itself as a global service available to the entire network, or limit its registration to the local system. Managers that choose the latter form of registration do not make themselves unavailable, but rather limit their visibility to clients that specifically probe their system for location information.

Clients find objects by querying the Location Broker for appropriate registrations. A client can choose to query for a specific object, type, interface, or any combination of these characteristics. When operations are externally constrained to occur at a specific location, a client can choose to query the location broker at the required system for managers supporting the appropriate object.

7.2 Location Broker Organization

The Location Broker is divided into two components. The *Global Location Database* is a replicated object containing the registration information of all globally registered managers; the processes that manage this database are called the *Global Location Broker*. The NCS runtime implementation of the Global Location Broker uses the *Data Replication Manager (DRM)* to maintain the database. DRM provides a weakly consistent replicated KSAM package. Weak consistency implies that replicas of the Global Location Database object may be inconsistent at any time, but, in the absence of updates, that all replicas will converge to a consistent state within a finite amount of time. This form of consistency provides a high degree of both read and update availability to the Global Location Database. It is not necessary to be able to communicate with all replicas of the object to affect a change in the registration database. The DRM assumes the responsibility of propagating updates to the replicas in a timely fashion.

A *Local Location Broker* supports managers that wish to limit their registration to the local system. Access to these registrations is provided in two ways. A client can directly query the Location Broker at specific node to determine the objects and managers that are registered there. Alternatively, a client can simply execute a remote operation while supplying an incompletely bound handle (i.e. one which specified only an object and system, not a particular server process). Remote calls made using such a handle are delivered to the Local Location Broker, which serves as a forwarding agent if an appropriate manager has registered itself locally. This mechanism obviates the need for users of the NCA to use well known ports.

The division of the Location Broker into two distinct entities is, to a large degree, an NCS runtime implementation decision. Logically the Local Location Database object and the Global Location Database object are a single partitioned object, and, in fact, access to these databases is provided through a common set of operations which select the target based on lookup keys.

8. The NCA/RPC Protocol and NCS Implementation

The NCA/RPC protocol is designed to be low cost for the common cases and independent of the underlying network protocols on top of which it is layered. The NCS runtime implementation of the NCA/RPC protocol is designed to be portable.

8.1 Protocol

The NCA/RPC protocol is designed so that a simple RPC call will result in as few network messages and have as little overhead as possible. It is well known that existing networking facilities designed to move long byte streams reliably (e.g. TCP/IP) are generally not well suited to being the underlying mechanism by which RPC runtimes exchange messages. The primary reason for this is that the cost of setting up a connection using such facilities and the associated maintenance of that connection is quite high. Such a cost might be acceptable if, say, a client were to make 100 calls to one server. However, we don't want to preclude the possibility of one client making a call to 100 servers in turn. In general, we expect the number of calls made from a particular client to a particular server to be relatively small. The reliable connection solution is also unacceptable from the server's perspective: A popular server may need to handle calls from hundreds of clients over a relatively short period of time (say 1-2 minutes). The server does not want to bear the cost of maintaining network connections to all those clients.

The well-known way of getting around the well-known problem of using reliable network connections is to make the RPC protocol implement exactly the reliability it needs on top of an *unreliable* network service (e.g. UDP/IP). This approach has the additional advantage that some systems

(e.g. embedded microprocessors) can not or do not support *any* reliable network service; however, if they're connected to a network at all, you can be sure that they'll at least supply an unreliable service. Further, unreliable services tend to be more similar across protocol suites than do reliable services. (For example, some reliable protocols might return errors immediately if the network partitions even though a virtual circuit is currently idle, while others might defer until the next time I/O is attempted.) This similarity means that the RPC protocol can be accurately implemented in more protocol suites than if it would be possible if it assumed a reliable service.

All that the NCA/RPC protocol assumes is an underlying unreliable network service. The protocol is robust in the face of lost, duplicated, and long-delayed messages, messages arriving out of order, and server crashes. When necessary, the protocol ensures that no call is ever executed more than once. (Calls may execute zero or one times and, in the face of network partitions or server crashes, the client may not know which.)

The NCA/RPC protocol operates roughly as follows. The client side sends a packet describing the call (a *request* packet) and waits for a response. The server side receives and dispatches the request for execution, and sends a packet in response that describes the results of executing the call (the *response* packet). If the client doesn't receive a response to a request within a particular amount of time, it can inquire about the status of the request by sending a *ping* packet. The server either sends back a *working* packet, indicating that execution of the request is in progress, or a *nocall* packet, which means that the request has been lost (or that the server has crashed and rebooted) and the client needs to resend it. The protocol gets slightly more complicated if the input or output arguments do not fit into one packet.

If a called procedure is non-idempotent, the protocol ensures that the server executes the call at most once. To detect old (duplicate) requests, the server keeps track of the sequence number of the previous request for each client with which it has communicated. However, the server considers this information to be discardable and it may discard it if it hasn't heard from the client in a while. (I.e. there is no permanent "connection" between the client and server.) Thus, it is possible for a long-delayed duplicate request to arrive after the server has discarded the information about the requesting client. To handle this case, the server *calls back* to the client (using an idempotent remote procedure call) to ask the client for the client's current sequence number. The server then uses the returned sequence number to validate the request. Note then that for calls to non-idempotent procedures (with input and output arguments that fit in a single packet), a total of two message pairs will be exchanged between client and server for the simple case. Subsequent calls between the same client and server will require just one message pair. Note that the extra message pair in the first case could conceivably be eliminated if the server were willing to hold onto client sequence number information for long enough to ensure that all duplicate requests had been flushed from the network. We chose not to take this approach since any time interval we considered long enough (e.g. one minute or more) seemed too long to oblige the server to hold the information.

Also, for non-idempotent procedures, the server side saves and periodically retransmits the response packet until the client side has acknowledged receipt of the response. If the server side receives a retransmission of the request, it resends the saved response instead of re-executing the call. The client side acknowledges the response either implicitly, by sending a new request, or explicitly, by sending an *acknowledgement* packet. The protocol also handles the case in which the server has executed the non-idempotent call but, because of network partitions or a server crash, fails to send the response packet.

If a called procedure is idempotent, the protocol makes no guarantees about how many times the procedure is executed. On idempotent requests, the server side does not save the results of the operation once it has sent back the response packet. In addition, the client side is not required to acknowledge the receipt of responses to idempotent requests.

8.2 Runtime

The NCS RPC runtime is written in portable C and uses the BSD Unix *socket* abstraction. (In terms of the socket abstraction, it uses SOCK_DGRAM-style sockets.) This abstraction is intended to mask the details of various *protocol families* so that one can write protocol-independent networking code. (A protocol family is a suite of related protocols; e.g. TCP and UDP are part of the

DoD IP protocol family; PEP and SPP are part of the Xerox NS protocol family.) In practice, however, the socket abstraction has to be extended in several ways to make it possible to write truly protocol-independent code. We extended the socket abstraction via a set of operations implemented in a user-mode subroutine library; the NCS runtime uses these extensions so that it can be truly protocol-independent. Bringing up the NCS runtime on a new protocol family should *not* require any changes to the NCS runtime proper. All that should be required is to add some relatively trivial routines to the socket abstraction extension library.

NCS is careful about creating sockets. Sockets are a fairly scarce resource and tying lots of them up for a long period is not a good idea. NCS keeps a small private pool of sockets. One is pulled from the pool when a process makes a remote call. When the call completes, the socket is returned to the pool. The pool need contain only one socket for the entire process if the system supports only one thread of control per process (as is the case in standard Unix).

The use of the socket abstraction at all could be considered to be too much of a BSD-ism, thus reducing the portability of the runtime. Fortunately, two factors argue against this point of view: First, it appears that AT&T System V, Release 3 will support at least a sufficient subset of the socket calls (layered on top of their own networking model). Second, even if the target of a port doesn't have anything resembling the socket interface, NCS use of the interface is fairly simple and it wouldn't be too hard to implement the BSD calls in terms of whatever the target system supplies.

9. Future Directions

NCA and NCS represent the first step in a complete network computing environment. One of the guiding goals in the development of NCA has been *transparency*. This has a number of aspects: replication, failure, concurrency, location, and name transparency.

With replication transparency all copies of an object can be considered equivalent. The user of an object cannot tell whether it consists of a single copy or many. The DRM provides replication transparency in the case where some short-lived inconsistencies can be tolerated. Future versions of NCA will include support for strongly consistent replication.

Location transparency allows users to access objects without specifying where the objects are. Objects are free to be moved around the network to adapt to changing load conditions and the availability of new hardware. The Location Broker provides the ability to find the location of objects prior to their first use. We would like to be able to have objects move at any time during program execution.

Concurrency transparency supports the illusion that a given client is the sole user of an object. NCS addresses this partially through concurrent programming support which provides a simple locking facility. In the future, we would like to address this, and to some degree, failure transparency, through the use of an object-oriented atomic transaction facility.

Failure transparency, i.e. the ability of components of a distributed system to fail and recover transparently to their users, is largely a function of location and replication transparency. By replicating objects, when a given replica fails another is available to take its place. Location transparency hides the switch from one replica to another from the user.

Neither NCA nor NCS address the issue of name transparency at this point. We anticipate building a general purpose name server in a future version of NCS. In addition, we intend to address a higher-level form of naming: In many instances, it is more convenient to find an object by attributes rather than by a text name. An *attribute broker* will provide this ability. Thus, a client will be able to query the attribute broker for a list of "26 page/sec laser printers" rather than managing the mapping between machine names and attributes itself.

Most of the focus in the NCA development so far has been on getting the basic model right. Once the object-oriented model is in place, we feel that these higher level services will evolve naturally. Had we started with a more traditional process-oriented model, the level of integration and transparency we desire would be much more difficult to achieve.

References

- [Almes 83]
Guy T. Almes. Integration and distribution in the Eden system. Technical Report 83-01-02, Department of Computer Science, University of Washington, 1983.
- [Birrell 84]
Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, II(1):39-59, 1984.
- [Cohen 75]
Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 141-160. ACM Special Interest Group on Operating Systems, 1975.
- [Goldberg 83]
Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [Lazowska 81]
Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth Symposium on Operating Systems Principles*, 148-159. ACM Special Interest Group on Operating Systems, 1981.
- [Leach 82]
Paul J. Leach, Bernard L. Stumpf, James A. Hamilton and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the Symposium on Principles of Distributed Computing*, 34-41. Association for Computing Machinery, 1982.
- [Leach 83]
Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, SAL-I(5):842-857, 1983.
- [Oppen 83]
D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems* I(3):230-253, 1983.
- [Stroustrup 86]
Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Sun 86]
Sun Microsystems. Networking on the Sun workstation. Part no. 800-1324-03. 1986.
- [Terry 84]
D. B. Terry, M. Painter, D. Riggle and S. Zhou. The Berkeley Internet Name Domain Server. In *Proceedings of the Usenix Association Summer Conference*, 21-31. 1984.
- [Wulf 75]
W. Wulf, R. Levin, C. Pierson. Overview of the Hydra operating system development. *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 122-131. ACM Special Interest Group on Operating Systems, 1975.
- [Xerox 81]
Xerox Corporation. Xerox System Integration Bulletin, OPD B018112. 1981.

APOLLO and DOMAIN are registered trademarks of Apollo Computer Inc. Network Computing System, NCS, and DOMAIN/IX are trademarks of Apollo Computer Inc.
IBM is a registered trademark of International Business Machines Corporation.
UNIX is a registered trademark of AT&T
XNS is a trademark, and ETHERNET is a registered trademark of Xerox Corporation.
VAX is a registered trademark of Digital Equipment Corporation.

SOLVING PERFORMANCE PROBLEMS ON A MULTIPROCESSOR UNIX SYSTEM

T. P. Lee
AT&T Bell Laboratories
Holmdel, New Jersey

*M. W. Luppi**
R. E. Menninger
AT&T Information Systems
Summit, New Jersey

ABSTRACT

This paper describes the kernel changes that solved performance problems encountered while measuring database benchmarks on a multiprocessor UNIX** system. Our concern was provoked by a surprising phenomenon: benchmark results on the dual-processor system were initially *lower* than on the uniprocessor. Monitoring of system activity revealed the causes of this fall-off: (i) the monopolization of the associated "slave" processor by low-priority compute-bound processes; and (ii) context-switch overhead from the formation of self-perpetuating queues, or "convoys," on high-usage semaphores.

The investigation yielded insights into the coordination of process scheduling between the two processors, and the proper management of key kernel semaphores. These issues are sometimes mentioned in the literature on a general level; we present here a case study, describing in detail the problems that arose and the applicable solutions.

1. INTRODUCTION

Multiprocessor versions of the UNIX operating system have become increasingly common in recent years. A primary issue facing the designers of these systems has been the coordination of processes simultaneously executing in the kernel on more than one processor. Careful management of this parallel activity is required to preserve the integrity of kernel resources. The original implementation of *sleep/wakeup* is sufficient to manage process concurrency for a multiprogramming environment on a uniprocessor [1], but it is not designed to handle real-time collisions between multiple processors accessing the same data structures.

One solution is to restrict kernel-mode execution to a single (master) processor (this is the method used in [2]). This solves the problem, but with a significant performance penalty: the master processor can end up with a disproportionate share of the load. (Kernel-mode execution can be typically forty to fifty percent of total system activity [3]). As the number of processors is increased, the master processor becomes a bottleneck, and the remaining processors become increasingly idle.

A different method was adopted for the implementation in [3], based on the use of semaphores to protect critical data structures. Processes on separate processors can execute concurrently in kernel mode; collisions between processes are resolved by semaphores similar to Dijkstra's P and V primitives [4]. This approach allows a more efficient distribution of the workload across processors, but introduces

* Author's current address is Morgan Stanley, 1251 Avenue of the Americas, New York, NY

** UNIX is a registered trademark of AT&T.

significant complexity into the design. It is a major task to determine (i) the appropriate granularity for semaphore locking; (ii) the proper placement of key semaphores; and (iii) effective coordination among processors when making scheduling decisions. Design choices yielding good results for most workloads can (as we found) lead to under-utilization of processors and a high level of lock contention for some common situations.

2. OVERVIEW

2.1. System Configuration

The multiprocessor described in this paper is a tightly-coupled dual-processor shared memory system, allowing processes to execute in kernel mode on either processor under semaphore control. The system belongs to what is commonly called the Associated Processor (AP) class of architectures [3], consisting of a "master" processor which manages all I/O activity (including I/O interrupts) and an associated "slave" processor. Processes initiating an I/O operation on the slave processor are preempted and subsequently rescheduled for the master. The system is otherwise fully symmetrical.

2.2. Initial Observations

We measured the initial version of the operating system for the multiprocessor with a broad mix of workloads. Most of our results confirmed earlier observations about the throughput gain provided by the dual processor (as much as seventy percent more than the uniprocessor [3]). One of our benchmarks, however, yielded a contrary result: throughput for the dual processor was forty percent less than for the uniprocessor. (These results are summarized in Table 1).

The benchmark consisted of a synthetic workload that mimicked common activities of a database environment, including the frequent creation, insertion, and deletion of records. The throughput index was based on the number of transactions accomplished over a fixed period of time. We also ran a number of compute-bound processes in the background (the total CPU consumption of these low-priority "cycle-soakers" helped us to estimate the true system idle time for all processors).

System instrumentation provided our first clues about the causes of the degradation. We observed the following:

1. *Unequal distribution of CPU usage among processors:* the slave CPU spent almost 100% of its time executing in user mode, while the master CPU distributed its time more evenly between user and kernel mode.
2. *Context switch overhead :* we observed a high rate of context switches per second, which suggested that semaphore contention could be involved.

Eventually we discovered two primary contributors to the degradation. One was the monopolization of the slave processor by CPU-bound processes, even when higher-priority processes were on the run queue. A second problem was the formation of long slowly-dissolving queues on critical semaphores. (The authors of this paper have borrowed the term "convoy" [5] to describe the observed queuing behavior). The remainder of this paper describes these problems in more detail, as well as the adopted solutions.

3. PROBLEMS AND SOLUTIONS

3.1. Isolation of the Slave Processor

We found that the slave processor did not contribute significantly to the benchmark index, since it was occupied mostly by the "cycle-soakers" that ran in parallel with the database workload. The underlying causes of this phenomenon were (i) the restriction of I/O activity to the master processor, and (ii) inadequate coordination of process scheduling between processors.

In the initial multiprocessor version, there was no global means for a processor to indicate when it made a process runnable. As a result, the other processor remained unaware of the change in the run-queue state, until its current process finished its time quantum or otherwise relinquished the CPU. A low-priority process could thus continue executing even when higher-priority processes were waiting.

This "isolation" was a particular problem for the slave processor, since it had no timely notification when the master processor unblocked a process after an I/O interrupt. As the following sections describe, the slave processor was therefore susceptible to domination by low-priority compute-bound processes that frequently used their full time quantum.

3.1.1. Processor Scheduling

In the original uniprocessor code, the system used a single state variable *runrun* to determine when the run queue should be examined for processor rescheduling. The variable was set when:

- i. The current process had used up its time slice, or
- ii. A process had become runnable as the result of an I/O interrupt or a *fork*.

The system checked *runrun* at each system-call exit or return from interrupt. If *runrun* was non-zero, the processor was immediately rescheduled. (If the processor was in kernel mode, rescheduling was delayed until system-call exit).

It should be noted that *runrun* had two distinct meanings: it denoted an event that was local to a single processor (the time slice had expired), and an event that was global to all processors (a process had been placed on the run queue). In the uniprocessor version, this ambiguity caused no problems. In the initial multiprocessor version, the system maintained a copy of *runrun* for each processor. The local meaning was preserved, but most of the global meaning was lost. A processor therefore had no way of knowing when the actions of another processor changed the state of the run queue. In particular, the slave processor was not immediately aware when the master placed a higher-priority process on the run queue after an I/O interrupt or a *fork*.

A low-priority compute-bound process on the slave was therefore never preempted until completion of its full one-second time slice. Although the slave always selected a higher-priority I/O-bound process over a compute-bound process, the former was usually soon rescheduled for the master, whereas the latter would effectively isolate the slave from actions of the scheduler for the full quantum. The net effect was that processes with compute-bound characteristics tended to drive off processes with I/O-bound characteristics, without regard to individual process priorities. As a result, we found that the benchmark's low-priority "cycle-soakers" consumed most of the slave CPU capacity, defeating the scheduler's priority scheme at the expense of the I/O-bound database workload.

3.1.2. Solution

Once we understood the problem, the solution was straightforward:

- i. we restricted the use of *runrun* to indicate expiration of the current time quantum, and
- ii. we included separate state information to track additions to the run queue.

When the state of the queue changes, the system now reschedules the processor at the next system-call exit, or (when in user mode) at the next return from interrupt. In the latter case, this means that rescheduling will occur no later than the next clock interrupt. This eliminates the original problem of slave-processor isolation.

We came away from this investigation with a stronger awareness of the need to distinguish adequately between "local" and "global" processor events. State variables should reflect this partitioning, in a way that provides each processor with timely information about state changes that affect scheduling decisions. Allowing each processor to proceed in isolation can undercut the ostensible goals of the scheduler, with consequent adverse effects on performance.

3.2. Semaphore Convoys

The isolation of the slave CPU meant that the master processor did most of the "real" benchmark work. One would consequently expect the throughput for the dual processor to be approximately equivalent to the uniprocessor. In fact, we found that performance was considerably lower on the multiprocessor. This further degradation was attributable to a second problem---context-switch overhead from "convoys" forming on critical semaphores.

The "convoy" phenomenon was originally described for database environments with a high rate of traffic on resource locks [5][6]. A convoy is defined as a stable queue that typically forms when a process holding a lock is preempted. The long holding time increases the chances of collisions between processes subsequently accessing the resource. The queue tends to be cyclically self-perpetuating, since it is likely (in a high-traffic situation) that a process will rejoin the queue shortly after releasing the lock. The overhead from context switching soon affects performance. One method for dissolving convoys is to disrupt the cycle; the suggestion in [6] is to grant the lock on a "random" basis, instead of FCFS.*

3.2.1. Convoy Behavior

During the benchmark runs, we observed severe degradation from convoys forming on key semaphores, in particular the *free-list* semaphore. This semaphore controls access to the list of free disk blocks associated with a filesystem. Since the database workload was constantly creating and truncating files, there was heavy contention for this semaphore. Normally the holding time for the semaphore was brief, since the tail-end of the free list, consisting of fifty free-block addresses, was kept in memory. It was generally sufficient for a process returning disk blocks (for a file truncation) or requesting blocks (for a write) to access the memory-resident list. Occasionally, however, an access to the disk-resident free list was necessary, when high demand had depleted the memory-resident list, or when a surplus of returned blocks had caused the list to overflow. (Figure 1 depicts the free-list structure). If a disk access occurred, the semaphore holding time was increased by a couple orders of magnitude.

One might expect that the allocation and the freeing of disk blocks would approximately balance each other, oscillating for the most part within the boundaries of the memory-resident list. For the database workload discussed in this paper, however, we found that depletions and overflows of the free list were frequent. This meant that the free-list semaphore was commonly held over long periods of time (from twenty to a hundred milliseconds).

In these circumstances, queues formed quickly on this semaphore, initiated by processes colliding when:

- i. a process held the semaphore during a synchronous disk *read* to obtain more free-block addresses, or
- ii. a process held the semaphore during a synchronous disk *write* to return surplus free-block addresses, or
- iii. processes on each processor attempted a near-simultaneous access of the semaphore.**

It should be noted that the first two circumstances can also affect a uniprocessor environment (and in fact we observed some convoy formation there). But the higher level of concurrent activity makes the multiprocessor more susceptible to this phenomenon, to the point where performance is severely affected.

Once a queue formed on the free-list semaphore, the queue length was quickly extended by new arrivals. (There were similar problems with the file-table semaphore as well). The observed characteristics can be summarized as follows:

1. *Creation*: Collisions between two or more processes initiated the queue. The collisions were most probable when the semaphore was held over a long interval (e.g., waiting for a disk transaction to complete), or when two or more processors were running. Both of these conditions were present on the multiprocessor.
2. *Perpetuation*: Once the queue formed, the window for further collisions was greatly increased. Each access then required a context switch, which effectively increased the semaphore holding time by half a millisecond for each process. Eventually, most of the processes could be found on the queue. At this point, every system call accessing the free list resulted in a context switch. As

* We conjecture that convoys were not previously observed on the uniprocessor because the earlier *sleep/wakeup* mechanism approximates a "random" allocation among all processes waiting for the resource.

** Once a queue formed, the average semaphore holding time for each process was increased by the context switch required to come off the queue. This made further collisions between processors even more likely.

long as the access rate remained constant, the queue was cyclically self-perpetuating, i.e., a process relinquishing the semaphore was again placed at the end of the queue when it attempted a subsequent access a short time later. The overhead of a context switch per access soon began to degrade performance.

3. *Dissolution*: Processes in competition for the resource inched along serially as long as the queue existed. The queue drained during intervals of low usage. The longer the queue, however, the greater its chances were of surviving a temporary dip in the access rate. If one or more processes were still queued when the rate increased, further collisions quickly restored the queue length. We concluded that it is therefore important to maximize the rate at which the semaphore queue is emptied.

3.2.2. Solutions

Solutions for eliminating convoy overhead fell into three categories.

1. *Minimizing windows during which queues can form*. Care was taken to minimize the holding time of semaphores in high demand. Our solutions here agree with one of the recommendations in [5], to avoid processor preemption wherever possible. Particular measures included:
 - i. *Avoiding synchronous disk accesses*. We changed the disk write that returned surplus free-list addresses from a synchronous write to an asynchronous write. (An asynchronous write allows the process holding the semaphore to proceed immediately, instead of sleeping until the write completes).
 - ii. *Replacing preemptive semaphores with spin locks*. We changed the file-table semaphore to a spin lock. This reduces the average semaphore holding time, since a context switch is not required: when two processes collide, the loser merely waits until the winner is done. Since the file-table access is brief, the CPU busy-wait is a minor overhead.
2. *Maximizing the rate of queue dissolution*. As was discussed previously, the queue on the free-list semaphore tended to be self-perpetuating. After relinquishing the semaphore, a process was usually able to run unimpeded until its next free-list access, when it would be added to the end of the queue. The solution was to alter the processor scheduling so that the queue empties before the relinquishing process could re-access the free list. In the revised design, a global counter is incremented when a semaphore is released at high priority. If the counter is active when the relinquishing process exits the system call, the process is preempted. It is not scheduled again until all higher-priority processes coming off the semaphore queue have a chance to run. The queue will usually be empty when the process again attempts to access the free list. We regard this solution as an aggressive alternative to the previously mentioned policy of "random" lock allocation [6].
3. *Maximizing the work accomplished within each semaphore call*. In the earlier implementation, the semaphore was accessed separately for each return of a individual block to the free list, even if the process was returning more than one block. This procedure worked well when there was no queue on the semaphore. If other processes were queued, however, the releasing process would promptly be added to the queue again if it had further blocks to release. This tended to keep the convoy alive. We changed the code so that a process released all blocks under one semaphore access.

The benchmark described in this paper exhibited a relatively homogeneous pattern of resource usage, which placed great pressure on the file-table and free-list semaphores. In environments with more heterogeneous resource usage, semaphore queues will appear less frequently. Internal simulations of semaphore-based multiprocessor designs have demonstrated, however, that as the number of processors increases, the likelihood of queue formation also increases [7]. Eventually, "lock thrashing" from semaphore contention has a noticeable effect on performance. Strategies that reduce semaphore holding time and maximize the rate of queue dissolution can substantially postpone the point at which this degradation begins.

4. SUMMARY

Initial measurements of a synthetic database workload gave an anomalous result: throughput for the dual processor was forty percent lower than for the uniprocessor. We found the following causes for this behavior:

1. Background "cycle-soakers" monopolized the slave processor, effectively preventing it from contributing to the throughput index.
2. Context switching from convoy formation was a much severer problem for the multiprocessor.

The situation called for changing the operating system algorithms, since we felt that this behavior was potentially typical of a wide range of workloads. (In particular, databases built on top of the file system were likely to be susceptible to the convoy problem). After implementing the changes, the throughput for the dual processor was approximately doubled. Slightly more than half of this gain was attributable to the scheduler modification that causes the preemption of low-priority processes on the slave processor. The balance came from the reduction of convoy formation. The latter solution not only helped the multiprocessor, but resulted in throughput improvements of more than ten percent for the uniprocessor. These results are summarized in Table 1; throughput gains are normalized relative to the initial single-processor results (prior to the kernel changes).

Table 1. Throughput (normalized relative to initial single-processor performance)		
Measurements	Dual Processor	Single Processor
Average results for previous workloads	1.7	1.0
Initial Measurements (database workload)	.6	1.0
With scheduling change	1.2	1.0
With scheduling and convoy changes	1.65	1.15

5. REFERENCES

- [1] K. Thompson, "UNIX Implementation," Bell Systems Technical Journal, Vol 57, No. 6, (July-August 1978), pp. 1931-46.
- [2] G. H. Goble and M. H. Marsh, "A Dual Processor VAX 11/780," Purdue University Technical Report, TR-EE 81-31, September 1981.
- [3] M. J. Bach and S. J. Buroff, "Multiprocessor UNIX Systems," AT&T Bell Laboratories Technical Journal, Oct. 1984, Vol 63, No. 8, Part 2, pp. 1733-1750.
- [4] E. W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, ed. F. Genuys, Academic Press, New York, NY, 1968.
- [5] J. Gray, "Notes on Data Base Operating Systems," IBM Research Report RJ2188(30001), February 23, 1978, p 83.
- [6] M. W. Blasgen, J. N. Gray, M. Mitoma, and T. G. Price, "The Convoy Phenomenon," IBM Research Report RJ2516 (May 1977; revised January 1979).
- [7] K. Siefkes, private communication.

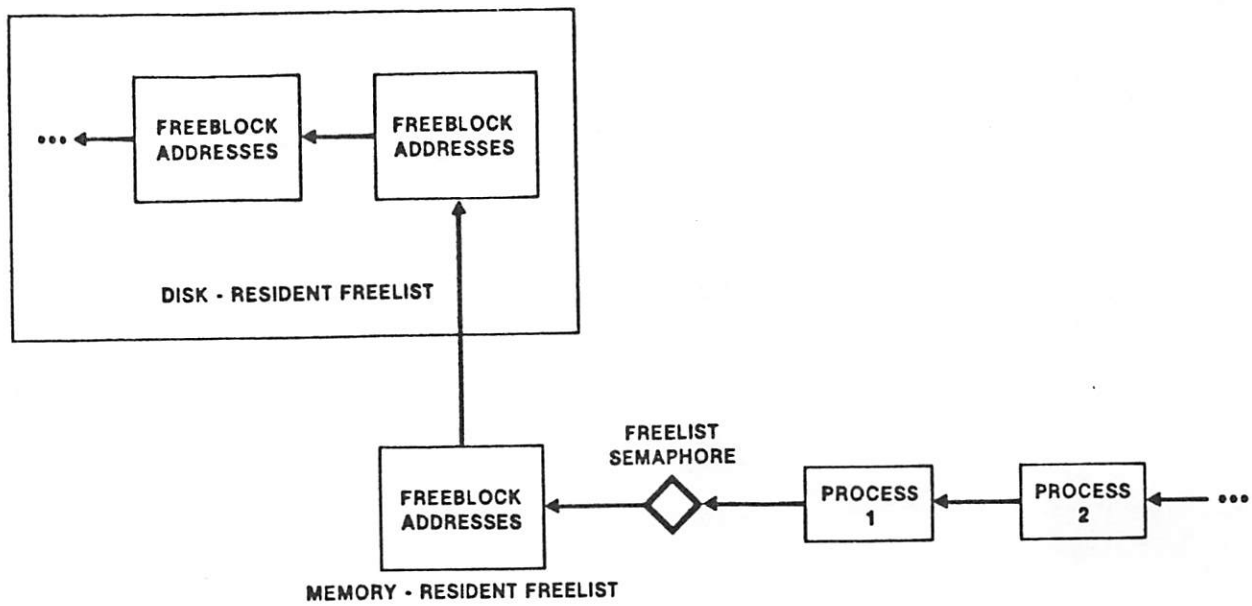


FIGURE 1 FILE SYSTEM'S FREELIST AND SEMAPHORE QUEUE

TAKING PERFORMANCE EVALUATION OUT OF THE “STONE” AGE

Ken J. McDonell[†]
Department of Computer Science
Monash University
AUSTRALIA
kenj@moncskermiit.oz

ABSTRACT

Predicting the performance of any computer system is critical to rational equipment acquisition by purchasers and successful product delivery by vendors. This paper takes a brief but critical look at “figure of merit” performance metrics, especially with respect to their reliability and predictive usefulness.

Necessary prerequisites for serious performance evaluation and prediction are identified. The architecture of the MUSBUS benchmark suite is described with particular emphasis on the technical considerations that allow MUSBUS to be tailored to accurately predict multi-user system behaviour in specific operational environments.

1. Performance Evaluation Goals

All computer system performance evaluation is directed to one or more of the following specific goals,

- (G-1) Compare the **measured** performance of heterogeneous systems executing specific identical tasks.
- (G-2) Compare the **anticipated** performance of heterogeneous systems executing the same “typical” tasks.
- (G-3) Collect diagnostic evidence to substantiate hypotheses about anomalous performance (either very good or very bad) for one or more tasks executed on a particular stable system.

Irrespective of the specific tests employed most performance metrics are based upon resource consumption (e.g. cpu time), throughput (e.g. as measured by elapsed time) or some related measure (e.g. number of users supported or mega-grunts per second).

Goal G-2 typically implies performance *prediction* based upon performance *measurement*, and this is by far the most useful application of performance evaluation. Purchasers want reliable estimates of expected system performance in *their* anticipated operational environment. This would be useful not only when new systems are being acquired, but also when upgrades are considered, e.g. “what demonstrable improvement can we expect from adding 2Mbytes of memory or a second disk controller?”. On the other hand, vendors need reliable estimates of system performance across a range of designated application environments to guide marketing and system tuning activities.

The central thesis of the first half of this paper is that G-2 is the most widely sought, but seldom realized, goal of current performance evaluation activity in the UNIX[‡] community. An approach to achieving this goal in any operational environment is described in the later sections.

[†] Currently on leave at the Department of Computer Science, University of Waterloo, Ontario, CANADA, kjmcdonell@er.waterloo.cdn or kjmcdonell@waterloo.csnet

[‡] UNIX is a Registered Trademark of AT&T.

2. Some Performance Tests and Measures

The UNIX system and the C programming language combine to provide a stable software execution environment on machines across the full price-performance spectrum. The consequent ease with which portable software can be developed has fostered a large class of tests purporting to measure system performance by a single “figure of merit” metric. These tests fall into two basic classes,

- (a) The “one liners” that are easy to type in, use standard UNIX programs and measure cpu time using either the shell built-in time function or `/bin/time`. Some common examples are shown in Figures 1 and 2.
- (b) Synthetic tests such as the “stone” family (whet[3], dhry[9,12], dhamp[5], ...).

Irrespective of which class they come from, these tests may be characterized as follows,

- (a) The test is easy to run, and is guaranteed to produce a value for the performance metric.
- (b) The value obtained is statistically unreliable due to an unknown and often large measurement error; worse still, the relative error may vary between different machines.
- (c) The performance metric typically measures some combination of
 - cpu arithmetic speed, and
 - C compiler quality.

Unfortunately, other important factors (as identified below) are totally ignored.

- (d) System performance under *real* operating loads may be, but most often is not, well correlated with the test and performance metric[6].

Clearly these tests potentially satisfy the performance evaluation goal G-1, and with careful use could assist with goal G-3. However the majority of people running and interpreting these tests appear to believe the results have some predictive value as per goal G-2. The compilation and publication of tabulated results from these tests under the heading of “UNIX benchmark results” is highly misleading because what has been measured is influenced by only a few of the many factors contributing to system performance – this is the fundamental weakness common to all the single “figure of merit” approaches. These test programs and performance evaluation suites are essentially of no more use than common sense and guesswork in predicting the performance of a *real* system under *actual* load conditions.

```
main()
{
    int    i;
    for (i = 0; i < 1000000; i++)
        ;
}
```

Figure 1: The “count to a million” test.

```
$ time dc
99k2vpq
1.414213562373095048801688724209698078569671875376948073176679737990732\
478462107038850387534327641572
          9.2 real          1.3 user          0.2 sys
```

Figure 2: The “square root of 2 to 99 decimal places” test.

To be fair, it is the use of the tests and interpretation of the results that is most commonly at fault. The tests have often been constructed for a specific purpose, and in that role are both accurate and useful. Despite the pleas of their creators (see for example [8,10,11]), and caveats in the code, it is the adoption of the test for an unsuited role (i.e. general performance prediction) that is the problem.

Another class of tests has evolved from recognition of specific areas of weakness in some UNIX implementations and/or factors perceived to be important influences on performance, for example

- filesystem throughput
- system call overhead
- fork() and exec() speed
- pipe throughput
- memory access bandwidth

These tests are clearly aimed at goal G-3, and any wider interpretation of their results cannot be made. Even in this restricted usage, these tests can be tricked by implementors aiming for a competitive edge in commonly used benchmarks (e.g. cache the results from getpid()). In some cases the tests are simply misguided, measuring behaviour that is uncommon in many operational UNIX environments (e.g. random file I/O).

Some attempts have been made to provide test environments in which many G-3 style tests are performed, and the results combined using relative weights of importance [1,4,7]. The disadvantages of this approach are,

- (a) the tests are not statistically independent (interaction between factors is not measured), and
- (b) accurate assignment of the weights of importance is more difficult than constructing a user-level workload profile as suggested below.

Finally, there have been some G-1 type tests developed for comparing heterogeneous systems executing the same set of end-user tasks, for example [1,2]. The predictive value of these tests depends upon the extent to which the supplied end-user tasks are representative of a particular operational environment.

3. Improving the Methodology

In the performance of various UNIX systems running the **same** task was accurately measured, the differences in the observed results may be attributed to some of the following factors,

- processor performance; includes raw speed, configuration options (e.g. FPU, data cache, co-processor) and interrupt servicing overheads
- disk subsystem performance; device characteristics, bus bandwidth and channel/controller configuration
- C compiler; the quality may vary dramatically with revision level
- UNIX kernel implementation; quality varies between base versions, ports and release levels
- filesystem configuration; allocation of filesystems across spindles and filesystem age
- real memory available to user processes
- configuration parameters; most notably disk buffer cache size and filesystem block size(s)

Reliable performance evaluation tests must be sensitive to all the above factors, because the performance delivered to the end-user can be seriously degraded by any one of these factors. The simplest test meeting this criterion is to measure the time required to perform some mixture of typical processing tasks from the anticipated operational environment. In this way, total system performance is measured directly, rather than attempting to isolate and measure the performance in each of the critical areas.

Whilst there undoubtedly exist classes of UNIX users with similar patterns of system usage, it is unrealistic to expect that one test or one mix of processing tasks will be representative for all operational environments. Rather, we should be aiming for tools that help us create, realistically execute and instrument the running of a representative collection of tasks on a variety of system configurations.

Serious performance evaluation requires,

- (a) Definition of the anticipated workload profile.
- (b) A test environment that will execute randomized tasks, chosen from the desired workload profile, for various levels of system load and record statistically sound measures of resource consumption. This test environment must be portable and extremely robust to maximize its usefulness and to encourage vendors to run user-specific benchmark tests, often at remote locations.
- (c) Careful documentation of the environment in which the test was conducted (hardware configuration, revision levels of the operating system and C compiler, workload profile, sysgen configuration parameters, filesystem partitioning, etc.).

A workload profile may be characterized by a set of independent user-level tasks, each typically corresponding to one or more program executions. For each task, the following information is required,

- the particular programs involved
- representative test data (data files, user input, patterns to search for, directory contents, etc.)
- relative frequency of execution

Identifying and describing the anticipated workload profile is a task of varying difficulty. In some environments, historical records (e.g. process or shell accounting) or known application usage provide accurate data from which the workload profile may be constructed. In other cases, informed guesswork is required.

For the MUSBUS multi-user test described below, a workload profile consists of an annotated shell¹ script with all associated data files. Tasks with high relative frequencies may appear more than once in the script.

4. MUSBUS

The Monash University Suite for Benchmarking UNIX Systems (MUSBUS) is a public-domain benchmark suite developed originally for equipment comparison during acquisition procedures.

The suite supports all three goals of performance evaluation with a simulated multi-user testbed facility and a battery of specific diagnostic tests.

The diagnostic tests have been designed to measure raw speed in very specific areas. Their execution is controlled by a shell script and parameterized so that the default values effecting test selection, size and duration may be over-ridden by command line options and environment variables. Table 1 provides a brief summary of these tests.

Of all the tests in MUSBUS, the simulated multi-user test is the by far the most complicated, most realistic and most likely to uncover operating system bugs. It is also the test specifically engineered to provide reliable predictions of anticipated performance (goal G-2) since it may be easily configured to perform "typical" tasks for any operational environment and then run on heterogeneous systems.

¹ The choice of "shell" is truly arbitrary, and may include any interactive application environment, e.g. an SQL database query language interface.

Test Name	Controlling Variables and Default Values	Description
arith	arithloop [1000]	A family of tests that compute the sum of a series of terms such that the arithmetic is unbiased towards operator type. Each major loop in the computation involves summing 100 terms; there are \$arithloop major loops. Repeated for all flavours of ints and floats.
dc		Compute the square root of 2 to 99 decimal places using <i>dc</i> . This test is due to John Lions (University of New South Wales) who has suggested it as a good first order measure of raw processor speed.
hanoi	ndisk [17]	A recursive solution to the classical Tower of Hanoi problem. \$ndisk provides a <i>list</i> of the number of disks for a <i>set</i> of problems.
syscall	ncall [4000]	Sit in a hard loop of \$ncall iterations, making 5 system calls (dup(0), close(i), getpid(), getuid() and umask(i)) per iteration.
pipe	io [2048]	One process (therefore no context switching) that writes and reads a 512 byte block along a pipe \$io times.
spawn	children [100]	Simply repeat \$children times; fork a copy of yourself and wait for the child process to exit.
execl	nexecl [100]	Perform \$nexecl execs using execl(). The program to be exec'd has been artificially expanded to a reasonable size.
context	switch [500]	Perform 2 x \$switch context switches, using pipes for synchronization. The test involves 2 processes connected via 2 pipes. One process writes then reads a 4-byte (descending) sequence number, while the other process reads then writes a sequence number.
C		Measure the time for each of cc -c cctest.c and cc cctest.o where cctest.c contains 124 lines of uninteresting C code (108 lines of real code after <i>cpp</i>).
seqmem	poke [100000] arrays [8 64 512]	These tests try to measure memory read accesses per real second. \$poke accesses are made into arrays of \$poke x 1024 ints. A cyclic sequential access pattern is used.
randmem		Like seqmem, but uses random access patterns.
fstime	blocks [62 125 250 500] where [.]	Sequential file write time, file read time and file copy time for files of \$blocks Kbytes. Temporary files will be created in the directory \$where. The <i>copy</i> time for the larger files is the best indicator of throughput and reflects the type of disk activity most commonly generated by compilers, editors, assemblers, etc.

Table 1: MUSBUS diagnostic tests.

Once a workload profile has been defined (as described in the previous section), several (typically 4) scripts are automatically created, each comprising a randomized permutation of all the tasks in the workload profile. A control file (*workload*) is also created to describe how each script should be run (refer to Figure 3).

The multi-user test simulates a variable number of users, each executing their own job stream. The job streams are chosen by cyclic selection from the scripts.

Control over the multi-user test rests with the program *makework* (refer to Figure 4) that performs the following functions.

- (a) Read the workload and script files into dynamically allocated buffers.
- (b) Make cloned copies of itself (via `fork()`) to run the job streams for groups of users (necessary due to per process open file limits).

```
/bin/sh -ie <script.1
/bin/sh -ie <script.2
/bin/sh -ie <script.3
/bin/sh -ie <script.4
```

Figure 3: Typical specifications for executing scripts (workload).

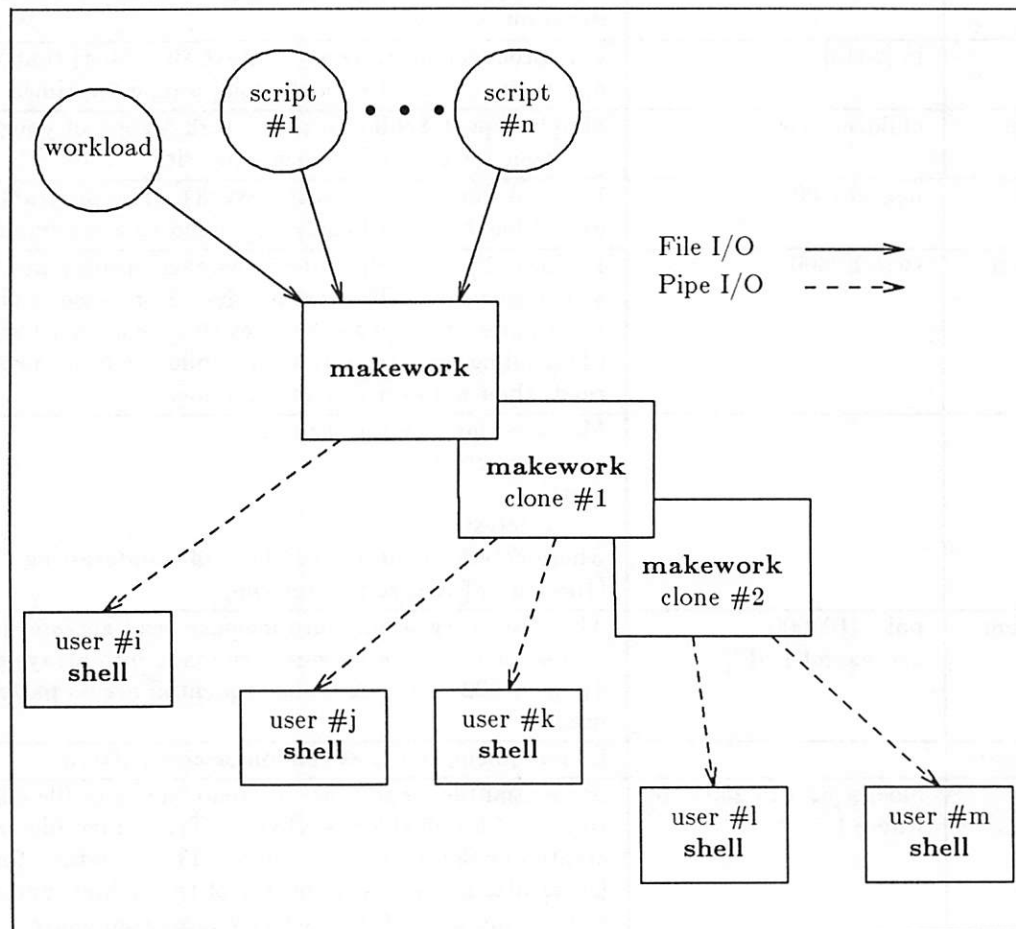


Figure 4: Overall architecture of the MUSBUS multi-user test.

- (c) Start each user shell with its input coming from *makework* via a pipe.
- (d) Send random chunks of input to the job streams, controlled so that the aggregate rate across all simulated users does not exceed a specified rate in characters per second.
- (e) All output from the shells and echoing of all input is directed to one or more real tty devices to ensure that an appropriate number tty output interrupts occur. See Figure 5.
- (f) When all script input has been sent, wait for all user shells and *makework* clones to terminate.

All MUSBUS tests are run under the control of a large Bourne shell procedure charged with.

- (a) Executing each test several times (the default is 6 or 3, depending on the particular test), recording the /bin/time results then computing the mean and standard deviation of the total (user plus system) cpu and elapsed times.
- (b) Reconfiguring the multi-user tests to allow tty and filesystem activity to be distributed across an arbitrary number of physical devices.
- (c) Performing tests with different control parameters, e.g. varying the number of job streams in the simulated multi-user test.
- (d) Monitoring completion status and standard error output to detect failed tests.

5. Issues Related to Test Engineering

The development of the MUSBUS multi-user test in particular has highlighted a number of issues related to benchmark test design in the UNIX environment.

Quiescent system configuration. Some tests must be run as super-user to avoid per user limits (e.g. maximum number of processes). However, given the performance prediction goals, the tests should be run on an otherwise unloaded machine in **multi-user** mode. This ensures that mandatory daemon activity will be present during test measurements.

Interactive input rate limitation. Limiting the *rate* at which input is presented to the shell and other interactive programs is an important factor influencing the predictive accuracy of the

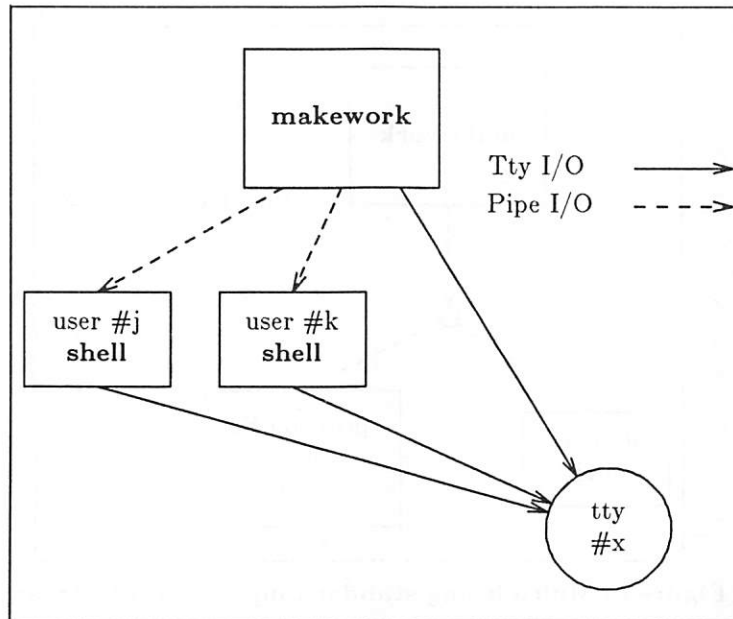


Figure 5: Directing terminal output to a physical device.

multi-user test results. Without this constraint, an interactive program's contribution to resource consumption for the job stream may be significantly reduced (e.g. artificially short program residency leads to higher buffer cache hits rates during `exec()` and application file I/O for repeated program executions, and reduced swapping and/or paging activity). Of course the ideal situation would be to simulate demand driven (i.e. no typeahead) and rate limited input. Unfortunately there is no portable and cheap (in terms of resource consumption) software technique² for one process to determine that another process is waiting for input, and so simulating demand driven input is not possible.

Bogus file sharing. If tasks in the job streams require access to the same data file, private copies should be made unless the files are truly shared in the application environment, otherwise buffer cache hits will artificially reduce the cost of file I/O. For example, if all N job streams contain an edit task on a sample data file, there should be N copies of the file made, one per simulated user.

Multiplexed standard input. When two processes compete in time for the one source of standard input (see Figure 6) serious problems may arise if the input generator (i.e. *makework*) is not response-driven. In general *makework* cannot tell whether the current "chunk" of input text is intended for the user's shell or some program invoked from that shell or a mixture of both. With reference to Figure 6 there are many pathological situations, the worst being program K consumes in one read some input that includes it's own termination command **and** some of the following text intended for the shell once program K has finished – the shell never sees that text! The architecture shown in Figure 7 has been used to overcome this; *keyb* includes the rate-limited text generation algorithm from *makework* and allows separation of shell and application input. However to retain control over the shell's rate of execution, the shell script must be padded with comments by the number of bytes in the input stream to program K.

Testing for failure. *Makework* checks the results of **every** system call, has a SIGPIPE handler and checks the status returned via `wait()`. If any error is detected, *makework* kills off all shells and the *makework* master kills off all clones (and their dependent shells). There is a certain degree of paranoia in this error checking, fostered by several bad experiences in which bizarre UNIX implementation bugs resulted in very good, but incorrect, predicted performance (if only a

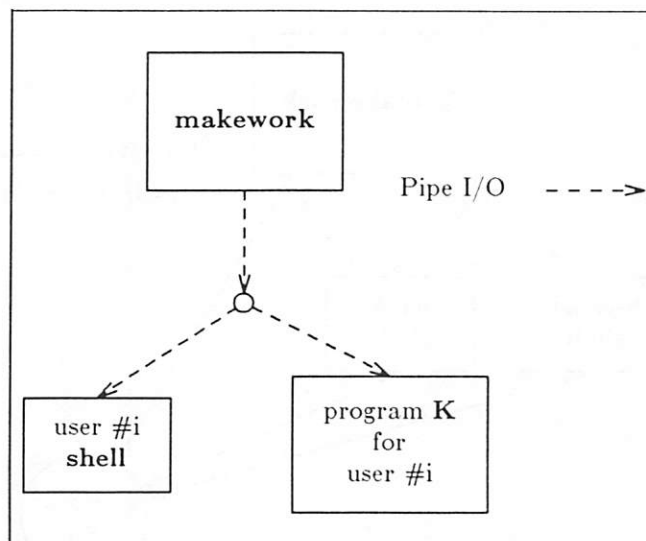


Figure 6: Multiplexing standard input in a job stream.

² Although external hardware "stimulators" have been used in some cases.

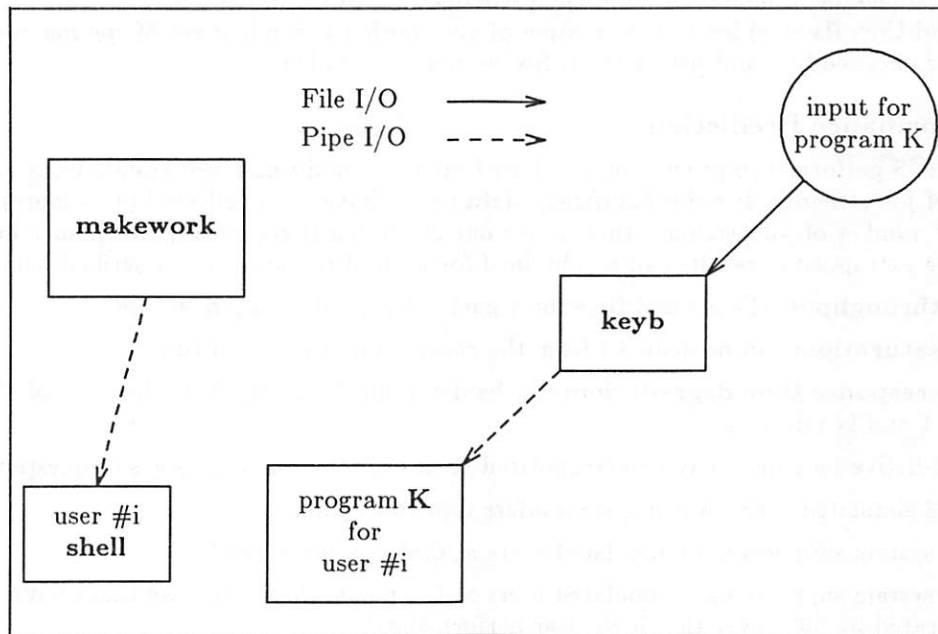


Figure 7: Multiple input sources for a job stream.

fraction of each job stream is executed, the work can be completed in a very short time!). No program or system call can be assumed to always execute correctly.

Randomizing the processing. Particularly misleading results are produced when a number of identical job streams are executed in effective synchrony. In some other benchmark suites, this has been used as a cheap way of increasing the “work” performed in a test. MUSBUS randomizes the processing load by using permuted scripts and randomizing the input rates to individual shells.

Using standard tools. MUSBUS uses many standard UNIX tools and utilities, in particular the Bourne shell, *awk*, *sed*, *grep* and a particularly bland vanilla dialect of C that is very portable. Uses include,

- permuting tasks to construct scripts
- checking standard error output for unexpected messages
- producing statistical summaries of repeated test results
- post-processing results to automatically produce *tbl* input for summary tables and tables comparing system performance
- the driving script, controlled by command line arguments and environment variables.

Constructing portable software. Since MUSBUS was specifically designed for comparing performance between heterogeneous UNIX systems, software portability was always an issue of importance. Despite the apparent uniformity of the C and UNIX interfaces, and considerable prior experience in building portable systems, a number of hidden incompatibilities were revealed in early MUSBUS usage. Some of the more notable problems included,

- */bin/time* produces different format output, which means different *awk* scripts are required to produce the statistical summaries.
- The behaviour of *wc* when given a single argument is not consistent.
- Trying to measure short elapsed time intervals varies between, impossible, hopelessly unreliable and grossly obscene code.

- The total lack of standardization in *c*pp predefined macros to reflect environment parameters (cpu and UNIX flavour) led to the creation of yet another redundant set of *c*pp macros that must be checked by hand before the software can be installed.

6. Performance Prediction

All MUSBUS performance predictions are based upon the multi-user test results for a varying number of job streams. Provided sufficient data points have been collected (4 or more) over a range of "number of job streams" that moves out of the linear region of performance behaviour, reasonable extrapolated results can be obtained for each of the measures described below.

System throughput: the elapsed time for a particular number of job streams.

System saturation: can be deduced from the ratio of cpu to elapsed times.

Relative response time degradation: can be determined directly from the ratio of elapsed times for 1 and N job streams.

These predictive measures may be extrapolated to answer the following sorts of questions.

- With 48 simulated users, which system offers best throughput?
- Which system supports most simulated users at 0.85 cpu utilization?
- Which system supports most simulated users at the point where response times have deteriorated by 50% over the single user performance?

Accurate interpretation of measured performance requires considerable skill and awareness of factors such as the following.

- (a) Particular hardware configurations, versions of the same Unix port and C compilers vary with time to such an extent that labelling one set of figures as from Brand X Model Y is misleading to all concerned.
- (b) MUSBUS is intended to be reconfigured in the multi-user simulated workload test to reflect the work profile of a particular user site. Whenever different workloads are used the results cannot be compared.
- (c) Deliberately the MUSBUS tests are in two distinct categories, raw speed and multi-user. The former are useful for diagnostic purposes only and give little useful information for a potential purchaser. The latter test gives good predictions of system performance.
- (d) Changing Unix configuration parameters (e.g. cache size, filesystem architecture, filesystem age, etc.) may have dramatic effects on the observed performance.
- (e) Beware of simulating **too few** users in the multi-user test. Useful information about system throughput and performance under heavy load conditions can usually be obtained by extrapolation of various measures computed from the CPU and elapsed times for the multi-user tests with various numbers of users. However this assumes the machine has been sufficiently loaded to move out of the *linear* part of the performance curves. For very fast machines, this may require emulation of a *large* number of users in the multi-user test.
- (f) Beware of simulating **too many** users in the multi-user test. This can result in unexpected resource depletion (e.g. serial line bandwidth) that does not accurately reflect the likely operating conditions.
- (g) Serious testing has been known to "break" UNIX ports. Causes have been identified as implementation (configuration) limits in the system being tested (e.g. proc slots), real bugs in the port or MUSBUS errors.

7. Concluding Comments

MUSBUS was originally developed to assist in equipment selection decisions. In that role it has proven to be most useful, and by empirical standards, an accurate predictive tool.

However the use has grown to include technical performance criteria to be met in contractual acceptance conditions, system check-out during installation, in-house performance measurement and kernel-exercising by several vendors during product evolution.

References

1. AIM Technology, AIM Benchmark Suite II – Evaluating UNIX Computers , 1984.
2. L. F. Cabrera, Benchmarking Unix – A Comparative Study, in *Experimental Computer Performance Evaluation* , D. Ferrari and M. Spadoni, (eds.), North-Holland, Amsterdam, 1981.
3. H. J. Curnow and B. A. Wichmann, A Synthetic Benchmark, *Comp. J.* 19 , 1, (Feb. 1976), 43-49.
4. G. Dronek, Relating Benchmarks to Performance Projections, *Proc. USENIX*, Salt Lake City, Utah, Jun., 1984.
5. R. J. Eickemeyer and J. H. Patel, Dhampstone, USENET, Newsgroup comp.arch, Article <500001@uicsg>, 14 Feb., 1987.
6. J. Mashey, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <112@winchester.mips.uucp>, 9 Feb., 1987.
7. M. F. Morris and P. F. Roth, *Computer Performance Evaluation – Tools and Techniques for Effective Analysis*, Van Nostrand Reinhold, New York, 1982.
8. J. H. Patel, Re: Dhrystone and Dhampstone, USENET, Newsgroup comp.arch, Article <500002@uicsg>, 26 Feb., 1987.
9. R. Richardson, Dhrystone, USENET, Newsgroup comp.arch, Article <153@homxb.uucp>, 14 Mar., 1987.
10. R. Richardson, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <2366@homxb.uucp>, 7 Feb., 1987.
11. R. Richardson, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <2387@homxb.uucp>, 14 Feb., 1987.
12. R. P. Weicker, Dhrystone: A Synthetic Systems Programming Benchmark, *Comm. ACM* 27 , 10, (Oct. 1984), 1013-1030.

The UNIX Marketplace in 1987: Life, the UNiverse, and Everything

Andrew Tannenbaum

Interactive Systems Corporation
Boston, MA 02116

ABSTRACT

I decided to write this paper while listening to the UNIX Retrospective session at the Winter 1987 USENIX in Washington, DC because I kept hearing the familiar tune that UNIX is elegant, small, portable, and wonderful. I mentioned during the question and answer session that sitting in the audience was like attending vacation bible camp – we all took a week away from home to listen while the leaders gave sermons to the faithful about life in the virtual world to come. We would go home and confront a reality that seemed only remotely like what we'd heard about at USENIX. This paper is about UNIX at home in that real world.

Pinning UNIX down

There are no recent papers about the essence of today's UNIX that stand out in my mind. I think it's because today's UNIX is not the pretty young thing that Thompson and Ritchie designed and wrote about. When you wrap an overgrown object in paper, you usually get an ugly package. Even if people have the guts to write papers about ugly subjects, the papers tend not to live comfortably in our memory.

AT&T doesn't allow the name UNIX to be used as a noun, only as an adjective, as in "the UNIX system." The reason for this is probably a legal one, involving trademark protection. However, it has another purpose. There are many different meanings to the word UNIX, so restricting the use of the label requires us to be more precise. That would make life easier, but people don't heed AT&T's warning when they discuss UNIX casually, since the law can't apply to every little conversation.

Defining UNIX is a most confusing task. Experienced programmers describe their feelings about UNIX when defining it. Users tell you about its user interface, or perhaps about their favorite UNIX applications. Vendors tell you about the distinct features of their products rather than about UNIX.

I can't quite define UNIX, but I know it when I see it. I'm a hacker and UNIX is my system, my familiar playground. As you become familiar with a tool, its capabilities become your capabilities. You become accustomed to it, and it disappears as an entity separate from you. Criticizing it becomes like criticizing yourself. It's easy to do, but the criticism is always biased.

UNIX is growing old

UNIX is not like it used to be, but I can still recognize it and I still love it. It has become heavy and sluggish with age, like Elvis Presley before he died. Too much money has been thrown at it; too many unscrupulous managers have tried to get their piece of the pie. It can still sing, though, and lots of down home folks think that it will never be surpassed. We look at UNIX with tears in our eyes, and imagine what it would have been like if it had had a different guardian angel. It might have developed into a most lovely creature, but more probably, it would have lived an uneventful life and died a quiet death.

UNIX is no mere mortal like you and me and, yes, Elvis. If it was, it would have already died of abuse. Not only is the mutant hulk not going to lie down and die, it is going to prosper in its own way, and many of us are going to prosper as we cause it to mutate. Computer software can take abuse that boggles the mind.

Architecture: How does UNIX develop?

The progenitors of UNIX used rational architectural ideas to create a well designed system. The process system, the I/O system, and the file system were models of simplicity. Ken Thompson, in his seminal paper on UNIX implementation, wrote:

What is or is not implemented in the kernel represents both a great responsibility and a great power. It is a soapbox platform on "the way things should be done." Even so, if "the way" is too radical, no one will follow it. Every important decision was weighed carefully. Throughout, simplicity has been substituted for efficiency. Complex algorithms are used only if their complexity can be localized.

Looking at the UNIX market today, you would guess that this philosophy no longer applies.

UNIX was originally designed using a process of piecemeal growth; someone (or some committee) didn't sit down and decide to design the operating system of the decade. A system can derive many benefits from being developed in small increments. Maybe most important, if you make mistakes, they're small mistakes. The problems tend to be easy enough for a person to fathom. Users can get involved in the design process, and have their individual needs addressed. Creative ideas have a greater chance of taking root – you're more likely to spend \$10,000 on an experiment than a million dollars. Small projects are more likely to integrate well with existing systems than large projects are.

UNIX was sheltered within Bell Labs Research for more than five years before its design was subject to the evil forces of the outside world. Creative progress was a by-product of casual experimentation, not of state-sponsored five-year plans or of promises made under duress. Today, engineering groups at many large UNIX companies work on very tight time schedules, and that work is usually fire fighting or radical redesign rather than evolutionary growth. Innovation takes a back seat to profit, or it finds no seat at all. Most companies rely on someone else to provide innovation – that's called cooperation.

Today's programmers and corporate planners in the UNIX game aren't well versed in the architectural language of form and fit. The bottom line is all important, and it doesn't matter if we have to twist UNIX back upon itself to accomplish our greedy end. In recent times, the effect of laying bare a forest of trees or of polluting a waterway has become plain to see, but we aren't as aware of the ecology of computer systems, the essential vitality of form that makes them successful, the form that can be strangled without proper attention.

The Industrial Revolution

There are proponents of various computer architectures who look down their noses at UNIX and say, "UNIX is no threat to us. We've sold billions of dollars worth of 370's (or VAXes or PC's or Apollo's) and we have the significant share of our market." Dinosaurs once had a significant share of their market. Horse drawn carriages did, too. These days, many American industries suffer the effects of market erosion because of poor planning. The manufacturing industries all suffered the pains of the industrial revolution in the 19th century, but the computer industry wasn't around at that time to learn the full lesson. Certainly, computer hardware and software are produced using modular designs, but most computer companies do not go far enough. Until UNIX, there has been no operating system plastic enough to accommodate a wide variety of other hardware and software application products.

Some vendors say that their systems already have entrenched user bases. Others say that their operating systems are better than UNIX – faster, cheaper, bigger, smaller, more reliable, more secure, more elegant, more sexy, more modern, more user-friendly, whatever. I will then ask them how many different hardware architectures it runs on, and they glibly reply, "Oh, VMS only runs on a million VAXes." Does VMS run on big systems? Well, it runs on a VAX 8800. Small systems? The MicroVAX. UNIX runs on Intel 80286 based systems and smaller. It runs on some of the biggest iron around, including IBM and Cray. It runs on new networked I/O architectures, multiprocessed CPU architectures, fault tolerant architectures. UNIX is an automatic first-choice operating system when a new CPU is designed – you will never see VMS run on any machine except a VAX.

The UNIX operating system is a manageable piece of software whose working source code is available to any hardware vendor, for less than the cost of one programmer/year! There is no comparable product available.

Portability vs Market Differentiation

Now every wise hardware vendor integrates UNIX into its product. You go to a trade show and you see rows of booths. In the booths are boxes with boards in them and boxes with screens and disks and tapes. Some boards are I/O controllers, usually the same from box to box, designed by controller vendors. Others are CPU boards, usually designed by the company whose name is on the box. The chips on the boards are usually the same chips from box to box. Same CPU's, same memories, same I/O busses. They're all made out of tacky tacky and they all look just the same. Most UNIX hardware manufacturers use off-the-shelf processor chips. They're proven safe and effective. The UNIX hardware manufacturer gets to ride the coattails of the chip manufacturer, easily incorporating advances in technology.

In a market where most products look alike, customers are hard pressed to choose one, and vendors are under even greater pressure to get customers to choose theirs. As in other competitive industries, the fight for market share boils down to the biased claims of marketing and sales forces, often under the guise of technical information. MIPS and Whetstone figures fly as furiously in our market as MPG and zero-to-sixty figures do in car advertisements, and they are of about as much value. You may be enticed by slick packaging or sexy demonstrations that may have little bearing on your use of the system, and the purchase price of a system probably ends up being a small fraction of the resources that you will eventually pump into it. It's hard to make the best decision; it's even hard to tell if you have made the best decision after the fact.

Allies and Enemies, Heroes and Villains

The UNIX market place is large enough to suffer from the effects of some interesting political issues. On the one hand, vendors woo customers by claiming that UNIX is portable and reliable. They purport to conform to industry standards, and of course their marketing and sales arms will usually make whatever claims the customers want to hear. But in the corporate boardrooms a conflict arises. While you'd like to attract customers with your portable system, you don't want the next vendor to come along with a more modern and effective portable system to steal away all your customers. So you have to design incompatibility into your UNIX box, and get your customers addicted to it. Vendors call this "value added." This might take the form of proprietary graphics packages, function libraries, or software packages that will only run on proprietary hardware, like user-friendly packages that require a certain array of function keys.

As vendors see the error of their similar-but-separate ways, they sometimes see the light and decide to cooperate on standards. There is a problem with the shift from proprietary architecture to standard architecture. Each vendor is married to its own ways, and each either wants its own way to become the standard, or, if it is not quite so entrenched, says "We'll accept any standard, as long as it's not our archenemy's." Standardization then becomes a schoolyard game rather than a cooperative technical pursuit.

#ifdef MADNESS:

**As UNIX becomes general-purpose on the outside,
it becomes hemorrhaged on the inside**

Market differentiation and portability are in constant battle. While each vendor must make its product attractive, the software often becomes increasingly mashed up. As vendors integrate their own support for multiprocessing, networking, internationalization, graphics, and databases, the kernel looks less and less familiar. Sometimes, these features are integrated with conditional compilation, and just as often, the #ifdefs aren't really conditional since the code won't work with them turned off. You end up with a maintenance nightmare that doesn't look like the base UNIX at all. When the vendors upgrade to the next standard revision of UNIX, they've got a long and tedious reintegration process to face. Some of us look upon this idea with horror, and others hear cash registers ringing.

Small was Beautiful, now we have Cycles to Burn

With the advent of greater and greater processing power, the question of how to use it best always has always arisen. I was weaned at Bell Labs, where the UNIX Powers That Be had some sense of UNIX aesthetics. Back in my hacker youth, I assumed that this reluctance to put features into the UNIX kernel and utilities was simply ignorance of what was going on outside the UNIX world. My requests from those

days would be considered mild today. I wanted to be able to suspend a running process. I wanted a screen editor. I wanted the machine to be able to feed characters to more than one terminal at 9600 bps. At the time, these features were thought to be inefficient. Computers were more expensive than people.

A screen editor would have to do massive amounts of screen update, and would have to take its input unbuffered. Suspending running processes would encourage people to use up memory with their swapped out stopped processes. In those days, one or two 9600 baud output lines would choke a PDP 11/70 or VAX 11/780 CPU. The power brokers said that if 110, 300, 1200, and 2400 bps weren't good enough for screens, then why would 9600 or 19200 be? Eventually, computers became cheap, people became expensive, and computers became people's tools rather than the other way around. Some people say that users will always complain about a lack of compute power. I think that this argument is a defense mechanism – better to call a user a whiner than to explain that we can't afford more power. These days, I don't hear as many complaints about lack of compute power as I used to, but I hear many ill-conceived ideas about what to do with the power. Many of these ideas entail making the machine less daunting to the novice user, since the potential market is made up of an unlimited supply of novice users. Attempts at making tools user-friendly are usually poorly engineered, like trying to design a foolproof chain saw. Radical changes to tools must be very carefully thought out.

The rat racers are not driven by a refined sense of aesthetics.

The UNIX command set is a well integrated tool for working on byte stream files. Filters and pipes provide excellent flexibility for working with files that contain plain ASCII text. UNIX programmers at one time were steeped in the practice of using existing tools to build new tools. They designed applications like database systems and text processing systems using existing tools or they designed new tools that integrated into the byte stream typeless file paradigm.

Unfortunately, not all forms of data can be easily represented in the byte stream, or often data can be more economically represented in other forms. Programmers who came to UNIX from other systems saw no crime in basing a WYSIWYG editor or CAD or database system on files that you can't cat or grep. When an engineer explains to a bean counter that, although the system will be 10% slower with ASCII data files, the 10% will pay itself back in simplicity, the bean counter insists that customers will never want to look at those files with cat, and that the competition doesn't have ASCII data files, and we need every bit of speed we can muster, so that we can kick tail in the benchmarks.

System V, Consider it Substandard

The UNIX market is currently suffering from a plethora of different versions. This is not a new problem in the UNIX world; UNIX has undergone a continual peristalsis since its popularity started to spread. The first bout of expansions and contractions was within the Bell System – originally, there were Research Versions One through Seven. From these Research Versions sprang UNIX versions controlled by the PWB Programmer's Workbench group, the USG UNIX Support Group, and by the CBUNIX group at Bell Labs in Columbus, OH. These three non-research UNIX versions converged into USDL System III just as Berkeley splintered Research Version Seven into the BSD branch of the tree.

With System V, AT&T advertised "System V, Consider it Standard." This is a vague suggestion, because standard is a vague term. In an advertisement, the implication is partly that it is a standard by which others are judged. There were certainly many users in the scientific community who favored BSD UNIX, because System V was not powerful enough to deal with current hardware innovations and user needs. In response to the BSD threat, System V's kernel has grown without bound in the past few years, incorporating a great deal of code to implement networking, file system sharing, and other features demanded by System V customers and BSD lovers alike.

Another meaning of standard applies even more strongly. While UNIX gurus like to see UNIX change with the times, UNIX users like to see UNIX sit still and keep working today like it did yesterday. Changes in UNIX often cause users to lose time getting their applications to run under the new versions. When AT&T told us to consider it standard, it was an implicit promise that users could rely on AT&T to be responsible about changing its function. To AT&T's credit, they have taken care of users and applications vendors by making most of their changes to System III and System V compatibly, more compatibly than Berkeley has made changes to BSD. While it might also be argued that some of Berkeley's

incompatibilities have come from their desire to track current technology, some of their frequent rewrites have become quite annoying – the signal system call implementation is a favorite member of the Hack of the Month Club.

AT&T still has control of the direction that System V takes, but while Berkeley still officially controls the BSD project, many of us think of the UNIX systems offered by Sun and DEC as synonymous with BSD. There are tens, perhaps hundreds of other vendors who have AT&T UNIX and Berkeley BSD source licenses, and each mutates the OS to its own taste. Most of the biggest vendors have undertaken the vast effort to integrate the AT&T UNIX and BSD systems, and there is a committee working on a portable standard operating system called POSIX. This standardization effort will surely cause another unifying contraction of versions within the next three years, and with this shift, and a POSIX committee will arbitrate the direction of standard UNIX growth, instead of having it dictated by AT&T and the other large UNIX vendors.

Cooperation in the UNIX community

AT&T and Berkeley distribute UNIX systems sources and then thousands of programmers at hundreds of companies end up doing the exact same work, fixing the same bugs, integrating the same two systems into one. If we were lucky, these programmers would be merely duplicating their efforts, but no, each company's programmers inject their own personal biases, creating hundreds of slightly incompatible mutants, each with rough edges to catch users unaware. There are some businesses where a duplication of effort is a necessary evil. It's impossible to design one house and send a distribution magtape to a hundred sites and just boot up new houses. The software business is well suited to benefit from cooperative efforts in quality assurance. Bugs don't just make computer systems work poorly, they cause applications programmers to install ugly workarounds in code that cost in efficiency and maintainability. While some companies might see a benefit in not sharing generic bug fixes, most would agree that they have better ways to spend their engineering resources than fixing the same bugs that everyone else fixes.

What kind of tool is UNIX?

In his Winter 1987 USENIX paper, John Mashey talks about UNIX as a lever. Most people think of a lever as a tool that provides leverage. Good enough. We should also remember that a lever is one of the simple machines – though I suppose that there are some who think of UNIX as more of a screw than a lever. Anyway, a simple machine is a device that can be used to make many different kinds of work easier, but it is not the tool that does everything. There are some people who wish they had one tool that did everything. It chops, it slices, it peels, it dices! It's a dessert topping AND a floor wax! As people cram more stuff into UNIX, they make it more unwieldy, turning it into a jack of all trades and a master of none. As UNIX gets bulkier, it becomes more expensive to port and maintain, and it becomes less attractive as the basis for a wide variety of products.

Where is the UNIX market today and where is it going?

UNIX has become another armored bandwagon upon which the gun-shy may ride. You never get fired for buying IBM, and these days you don't get fired for suggesting that your new computer product have UNIX built in. UNIX wasn't, isn't, and will never be the best answer to every need for an operating system.

In a paper he presented at the Winter 1984 UNIFORM, Brian Redman wrote:

Ergonomic designs don't seem to take my needs into account. I don't want a user-friendly system. I'm not a friendly user and neither are my colleagues. We're inconsiderate ogres without the slightest regard for the machine. We expect it to respond on command, to work endlessly and not to put up a fuss. Rather like a mute slave who's only purpose is to silently obey.

UNIX was designed using minimalist principles. Some people have complained that it is too terse, but I think that therein lies its charm. Like salt in soup, it's easy to add features to UNIX, but it's hard to remove them from other systems. We love UNIX for what it isn't, because that makes it easier for us to make it what we want.

I can safely predict that UNIX will run bigger, faster, and cheaper in the future. The UNIX system's flexibility makes it a rapidly evolving organism that is subject to mutation. As the internal injuries to UNIX have become malignant, people have redesigned its organs to better cope with the modern world. Where UNIX goes all depends on where individuals decide to take it. As with any other evolutionary process, I believe that the changes which are gradual, positive, and non-invasive are most likely to survive.

Suggested reading

Christopher Alexander, *Center for Environmental Structure Series*, Oxford Press.

Alexander has written a series of books that address architecture from an organic systems-design perspective. His ideas apply well to the problem of breaking a complex system down into manageable components. He discusses piecemeal growth in *The Oregon Experiment*, 1975.

Jon Bentley, *Programming Pearls*, Addison-Wesley, 1986.

Using well chosen examples, Bentley shows how to attack the heart of an engineering problem.

Ken Thompson, *UNIX Implementation*, Bell System Technical Journal, July-August 1978.

This BSTJ is the definitive work on the UNIX design.

UNIX at the Turn of the Century

*Michael Tilson
HCR Corporation
130 Bloor Street West, 10th Floor
Toronto, Ontario M5S 1N5 Canada
(416) 922-1937
{utzoo,ihnp4,...}@hcr!mike*

ABSTRACT

UNIX has been available outside Bell Labs since about 1974. Thirteen years ago the system was new, still experimental, and rarely used. Today, UNIX is mature, becoming standardized, and widely used. What can we expect in the next thirteen years? This paper discusses the technology trends that will determine the status of UNIX at the turn of the century.¹ Consideration is also given to the perils of forecasting.

Introduction

UNIX has become a standard working environment for software development. The lifetime of standards is surprisingly long. FORTRAN has been with us for a long time, and it looks like it will be with us for decades to come. Today's UNIX system will still work fine until at least late January, 2038.²

On the other hand, technology continues to advance at a rapid rate. Systems that once appeared modern become obsolete and obstacles to productivity. There is no reason to believe that the rate of change will slow between now and the end of the century. The important trends that must be considered include memory sizes, processor speed, network bandwidth, networking and communications software, user interface hardware and software, and software development technologies. We will see low cost, extremely powerful, more productive computer systems, that have very high bandwidth connections to other systems. UNIX must adapt to these changes.

Thirteen years ago the UNIX system was new, still experimental, and rarely used. Today the system is mature, widely used, and becoming standardized. The existence of virtually identical software environments on almost all machine architectures opens up possibilities that never before existed. The multi-vendor networked file system demos that now occur at many UNIX commercial exhibitions would have been unthinkable not very long ago.

¹ The pedantic reader will notice that the turn of the century is assumed to be the year 2000, and of course this really happens January 1, 2001. However, I suspect that when the time comes, the big celebration (or the wait for the end of the world) will come a year earlier. Anyway, UNIX programmers prefer 0-indexing.

² On 32-bit processors the current UNIX time algorithms will overflow after this date. Still, this is quite a bit better than some other systems that will fall over dead after December 31, 1999. When 64-bit processors become the norm, future timekeeping may be restricted only by limitations of storage needed to hold the time zone and daylight savings algorithms.

In the next thirteen years UNIX will open the door to possibilities for distributed processing and distributed applications that go far beyond anything we can do today. In this paper I attempt to reconcile the conflict between the pressure for change and the inertia of standards.

A technical forecast is provided, giving a framework for looking at UNIX systems development over the next decade. The goal is to understand why a typical obsolete C application written in the mid-80's might be still running on an incredibly advanced architecture, moving data from New York to Tokyo in the year 2000.

Forecasting

Forecasting for the next millennium is a dangerous business. Many people believe that the advent of a new millennium triggered a plethora of forecasts. In fact, the year 1000 was not considered terribly special by people at the time. Giving mystical significance to round numbers (e.g. 0x1000) is a more modern innovation.

However, the 11th century did mark the early beginning of a tradition of millenarian prophecy that persisted for centuries.³ These prophecies were not notably accurate, but they did trigger upheaval, crusades, and the like. Certain UNIX market forecasts of a few years back had a similar effect.

Today prophecies are a business. In our mercantile society we make predictions about markets and technology. In this paper I will not attempt to make a precisely accurate analysis of technical progress. I am more interested in order-of-magnitude effects. If a new age is dawning — or the world is coming to an end — that is of greater significance than whether the precise date is 2001 or 2002. (Besides, it's embarrassing to stand on a mountain top waiting for the arrival of the heavenly hosts if they don't show up on schedule.)

Unlike prophecies of the coming Millennium, I believe we will see a surprising inertia in our computer systems despite rather dramatic changes.



Venture capitalists scourging the forecasters. Will the turn of the century see the Day of Wrath for UNIX systems, can we expect the UNIX Millennium, or is the answer somewhere in between?

³ The "Millennium" is *not* the period commencing with 1001 A.D. It is supposed to be the thousand year reign between the Second Coming and the Final Judgement referred to in the Book of Revelation.

Hardware Technology Trends

The biggest advances will be in hardware. Fred Brooks recently noted that our software ability is not advancing at anywhere near the same pace, and this differential in the rates of progress does not seem likely to change. He suggested that there are fairly fundamental limits on the rate of improvement in software. Change happens, but in a more incremental fashion.

This is not to say that we won't see big improvements — we will. Nevertheless, the advance in hardware and communications technology will be much faster, and we need to understand that first.

We are still orders of magnitude away from ultimate physical limits on computer technology. For mid-range computers there is no reason to believe that these limits will be significant at the turn of the century. Mid-range computers for technical and business professionals are the most common "UNIX machine". Ultimate limits will first be reached on the "big iron", but given a workable computing technology, there is no doubt that it can eventually be delivered cheaply after enough time passes.

In 1974, a typical low cost UNIX machine was a DEC PDP-11/40, with perhaps 128 Kb of main storage, 0.3-0.4 MIP speed, 16-bit memory, 10 Mb of disk storage, a 9-track tape for bulk storage and information transfer. Both local and long-haul "networking" were provided by 110 baud modems.⁴ Such a machine in a usable configuration cost on the order of \$85,000. In 1987 dollars, the cost would be equivalent to about \$200,000.⁵

Some Hardware Trends

In this hardware forecast, you will notice certain assumptions, including an assumption of fairly conventional architectures for both hardware and software. Justification for the assumption will follow.

We should look at the history of computer technology in the last 13 years. In every respect, the hardware has vastly improved in performance and rapidly dropped in price. The improvement rate has been exponential.

The following table gives some comparisons, for machines that are typical small (but more than minimal) commercial UNIX systems. I am considering machines that can be purchased off-the-shelf, and that are available reliably and in quantity from established vendors. (The prices are intended to represent "usable" configurations — with wheels and engine. A stripped-down machine with limited or no disk storage, or a bare processor box can of course be obtained for much less, but it is more useful to compare real standalone configurations that could be used as complete systems for production work.) In the table, "User Presentation Rate" refers to a guess at effective net rate that information can be presented to the user. It is not a pixel update rate, which would be at least a factor of 10 higher.

⁴ I installed Fifth Edition UNIX in early 1975 on exactly such a system. At the time, it seemed like quite a capable system. The machine had a GT-40 graphics processor, which could be used under UNIX with a special driver. Unfortunately, such a powerful and expensive machine couldn't be used by only a single user, and timesharing the computer made interactive graphics somewhat difficult.

⁵ Inflation rates derived from figures published in *Monthly Labor Review*, March 1987.

"UNIX Machine" Performance and Price

	1974	1987
Speed (MIPs)	0.35	3.0
Bus width (bits)	16	32
Main memory (Mb)	0.15	2.5
Mass storage (Mb)	10	100
Avg. disk access time (msec)	70	15
Communications speed (Kbit/sec)	0.1	2.4
Local network speed (Mbit/sec)	N/A	1.0
User presentation rate (Kbit/sec)	10	10-100+
System price (1987 US\$)	\$200,000	\$20,000

A modern UNIX machine might be typified by a 68020-based workstation (e.g. Sun) or perhaps a 386-based machine. For approximately one tenth the price (corrected for inflation), today you can obtain a machine with about 10 times the performance, or an improvement in the total performance/price ratio by a factor of 100.

It is interesting to ask "Why don't our machines 'feel' 100 times better?" Part of the answer is in the software, which we'll discuss in a moment. Also, certain mechanical parts (such as disk drives) provide bottlenecks that have not improved at the same rate. Finally, some of the performance improvements have been devoted to user-interface issues. Productivity goes up, but not necessarily in direct proportion to the MIPs and megabytes devoted to window management, etc. Nevertheless, if you suddenly had to go back to sharing a PDP-11/40 with eight users, you would certainly realize that we have come a long way in a short time. (Back then, using a screen editor was considered anti-social, because of the load it placed on the system.)

UNIX Machines at the Turn of the Century

Again, there is little reason to believe that the current small and mid-range computers are pushing any fundamental technical limits. In the next thirteen years, we can expect similar trends to apply.

A Typical Low-cost UNIX Machine, 2000 A.D.

Speed:	25 MIPs
Bus width:	64 bits
Main memory:	32 Mb
Mass storage:	1000 Mb
Avg. 'disk' access time:	5 msec
Communications speed:	50 Kbaud (ISDN)
Local network speed:	10 Mbit/sec (effective rate)
User presentation rate:	1 Mbit/sec
System price (1987\$):	\$2500

The above is the result of applying less than another factor of 100 improvement.

Is this realistic? We should look at each factor separately in more detail.

Processor Speed

Today, microprocessor chips are approaching 10 MIPs. It will certainly be possible to build a small, cheap computer with 25 MIPs speed by the turn of the century.

Bus Width

The need for speed will motivate an increase in bus width. Also, increased requirements for virtual memory space and greater use of mapped files or "flat store" techniques will eventually cause us to make the jump to 64-bit processors. (This is starting to happen already on mainframe computers, where 32-bit addressing is running out of steam.)

Main Memory

By the turn of the century, 16 Mbit memory chips should be mature. The 1 Mbit chips are now starting to reach the market. If we allow four years for each generation to mature, then these will be fully mature by 1991, the 4 Mbit chips by 1995, and the 16 Mbit chips by 1999. We may be surprised by a faster rate of progress, or delayed by the technical problems of miniaturization. Previous memory chip generations have advanced a bit more quickly, so this forecast seems reasonably conservative. Not counting error-correction, a 32 Mbyte memory would consist of 16 memory chips, so this amount of memory should be easily available on a low cost system.

Mass Storage

Workstation storage capacities are already coming within this range, so this part of the prediction seems relatively safe.

Average Access Time

Continued incremental advances in servo control technology should bring 5 msec access times within easy reach. For this memory capacity at low cost, it is likely that rotating mechanical elements will still be involved, so there is a limit to the rate of improvement in access time.

Long Haul Communications

Dial-up voice lines can't carry data at rates much beyond 10 Kbit/sec. But I assume that ISDN (Integrated Services Digital Networks) will finally be available by the turn of the century. This means that 50 Kbit/sec digital connections will be available as part of the telephone system, and direct computer-computer links could be set up as easily as today we can establish modem connections. The interface should require only one or two chips.

Private corporate networks could be faster. However, an interesting part of the UNIX phenomenon has been the linkage of UNIX systems on demand according to immediate user requirements by way of UUCP and the dial-up phone network. ISDN will provide the same flexibility at lower cost and higher data rate.

LAN Speed

The technology will continue to advance. 100 Mbit/sec or faster networks will be in use, and after the software is done throwing away the usual 90%, we should gain at least a factor of 10 in performance of low-end LAN networks at reduced cost. The interfacing will be handled by a few chips.

User Presentation Rate

Video display technology will continue to advance. The amount of information that can be displayed depends on mass production of precision display devices. The advent of high definition TV means these will be a consumer item at low cost. Everything else simply depends on memory and MIP rate.

Price

Given the expected advances in processor and memory technology, the computer I have described is quite conventional. It could certainly occupy the equivalent of a single board, and it will use mature technology, so the cost should be low. Every piece of the puzzle exists today, so it is simply a matter of moving down the miniaturization/mass production/cost learning curve. There is simply no reason not to expect such cost reductions, at least to the nearest order of magnitude.

The High Performance Workstation

While it is exciting to look at advances in low cost PC-class machines we should also look at the more expensive professional workstation. What happens if we apply a similar performance/price improvement?

A Typical UNIX Workstation, 2000 A.D.

Speed:	50 MIPs
Bus width:	64 bits
Main memory:	250 Mb
Mass storage:	10 Gb
Avg. 'disk' access time:	4 msec
Communications speed:	50 Kbaud (ISDN)
Local network speed:	100 Mbit/sec (effective rate)
User presentation rate:	100 Mbit/sec
System price (1987\$):	\$25,000

Other features: Video input
 4K x 4K full-color screen
 Image (still or motion) and voice processing
 Rapid access to very large archival databases
 Expert systems components
 Use of publication quality color printing

As usual, the very cheapest hardware represents the best ratio of performance to price, but this workstation is still very competitive for the price. Even though this machine costs ten times as much, it isn't reasonable to expect 10 times the MIP rate. Similarly, there is no reason to assume a wider bus. The professional user will need to process larger amounts of information, and it can be assumed that much of this information will be in the form of images, diagrams, or databases; therefore larger amounts of storage are required, both for main memory and for mass storage. To fit in an office machine, main memory may require higher density chips (and perhaps be more costly). Access times will still be in the same ballpark as the smaller machines. Commercial LANs will use fiber optics, so higher data rates can be assumed (although no single workstation will consume the full fiber optic bandwidth.)

Is This Really Possible? — A Test

When I was first installing UNIX on a PDP-11/40, there was an IBM System/370 main-frame running in the same building. This machine was a multi-million dollar item. The upgrade for virtual memory alone ("the DAT box") cost several hundred thousand dollars. It had over a megabyte of main memory (!), several hundred megabytes of "DASD", and a processor speed of several MIPs. In other words, *it looked a lot like a typical high performance workstation*, except it was hard to use interactively and it took several rooms to hold the system components.

One test of this forecast for a high performance workstation is: Today can you buy roughly the same performance in a mainframe computer for 150 times the forecast price? (We are assuming a 100 times improvement in performance/price, and also that at any given time mainframes are at least 50% worse in raw performance/price ratio compared to small machines.) Well, for less than \$4,000,000, I think today you could buy a configuration from a number of manufacturers that has approximately the same basic characteristics as shown for the year 2000 workstation.

What Isn't Going to Happen?

Arthur C. Clarke once said that when a scientist claims something is possible, it probably is, but when a scientist says something is impossible, you don't really know. Since I'm not a scientist, maybe I can get away with some statements about what won't happen.

With respect to the large scale use of "affordable" computers, I don't think we'll see any of the following by the turn of the century:

- Widespread use of non-Von Neumann architectures
- Widespread general-purpose use of massive parallelism
- True artificial intelligence or "Fifth Generation Computing"
- Widespread use of provably correct programs in complex applications
- Revolutionary change in the way software is produced
- Replacement of UNIX as popular standard

Many people think that computing will be dramatically changed by radical new architectures. I believe there are strong economic obstacles that will prevent such change by the turn of the century. We have not seen the end of the learning curve with our existing technology. Today's conventional mainframe will become tomorrow's desktop computer. The pace of improvement is so fast that it will overwhelm the performance improvements possible with new architectures. New architectures are expensive, since they are at the start of the learning curve. They are especially expensive because we don't understand how to build software that makes good use of them.

If we see any success with unconventional new architectures, it will be in the supercomputing or mainframe areas, where we may be exploring fundamental technology limits. The first "teraflop" machine may require new methods. The first desktop supercomputer won't require anything *fundamentally* new. We already know how to build supercomputers. We just have to make them cheaper and smaller. This seems inevitable.

Even ordinary computing will consume the exponential growth in hardware technology with ease. True artificial intelligence seems to require unbounded resources. I feel safe in predicting that a mere factor of 100 improvement will not solve the problems in AI, and progress will continue to be disappointing. The complexity of our software tasks will also continue to frustrate the vision of provable programs.

We will continue to make incremental improvements in how we produce software and how we use it. Compared to 13 years ago, perhaps software technicians are twice as productive. We may make another similar gain by the turn of the century. It's possible that computing technology may eventually increase white-collar productivity in general, however to date it hasn't really happened. (All those PC's with spreadsheets have turned a lot of managers into programmers, but it isn't clear that we need fewer office workers today to carry out the same tasks than we did in the last decade.)

What About the Software, and Where Does UNIX Fit In?

April 1987 saw the 30th anniversary of the delivery of the first production FORTRAN compiler, for the IBM 704 (a vacuum tube machine.) Today, FORTRAN is standard, and still widely used for new software development. Each computer vendor conforms to a base standard (which is required for government bidding) and each vendor also supplies incompatible proprietary extensions. FORTRAN is recognized as having problems, but also is recognized as a good vehicle for achieving software portability. Standards committees are actively developing small modifications to fix deficiencies in the language. Does any of this sound familiar?

As I have already suggested, software carries greater inertia. Because of the complexity and interconnectedness of the tasks we carry out with software, it may be that we require exponential improvements in hardware in order to achieve linear improvements in overall productivity or usability. This inertia means that prospects are bright for UNIX at the turn of the century.

UNIX is only reaching the FORTRAN-66 stage — we are about to have our first formal standards. As a true industry standard, UNIX is still relatively young. (After all, UNIX as a *commercial product* is really less than a decade old.) With hardware advancing and changing so quickly, UNIX becomes the only defense for users. This standardization will drive the market for UNIX as the universal glue of the computer industry.

Proprietary systems like DEC's VMS or IBM's OS/2 will have UNIX layers or UNIX compatibility. Alternatively, UNIX systems will provide VMS, OS/2, or MS DOS virtual partitions for compatibility as needed.

Distributed Systems

The early UNIX tool philosophy meant that the system didn't make distinctions that weren't necessary: files are one type, devices look like files, a connection to a program via a pipeline looks like connection to file, and one machine looks much like another because of portability. This concept continues today with networked file systems: you shouldn't be able to tell where one machine ends and the next begins, because for most applications it isn't important.

With UNIX in widespread use, we will finally have an opportunity to solve problems in connectivity, communications, and distributed applications. Machines perform various tasks with greater or lesser efficiency and we want to partition our applications to use the right machine for the right task. While interaction is best handled by smaller machines, where cheap MIPs can be devoted to response time, aerodynamics calculations may be most efficient on the largest possible machine. Corporate databases need central administration and security. It is now possible to link all of these together with one applications interface and one uniform network, using UNIX.

UNIX will be a vehicle by which distributed applications and transparent networking of diverse resources will consume the cheap MIPs of the future.

This transparent diversity represents one of the most exciting prospects for UNIX systems, but it also raises some concerns. If UNIX really does proliferate, at some interface level it will become almost impossible to change. When our programs blindly copy files from Tokyo to New York over a networked file system, then neither the Tokyo nor the New York office will change the interface without great cost and risk.

Performance Range

UNIX is the only system that currently spans the performance range we are discussing. Today, UNIX is supporting "user-friendly" desktop applications and it is also supporting supercomputer applications. When we suddenly have desktop supercomputers, where will we get the software? Development of reliable, efficient, and well-conceived systems software takes a long time. Will MS DOS or OS/2 do the job? How about MVS or some of the batch-oriented supercomputer systems?

In terms of software development cycles, desktop supercomputers will be here tomorrow. Only UNIX, for all its flaws, comes close to being ready.

Opportunities for the Turn of the Century

Given the frantic pace of hardware development and the more leisurely rate of change with software, what opportunities do we in the UNIX community have?

Linear Improvements

There is always the opportunity for more of the same. We can burn MIPS and memory to make linear improvements. Since MIPS and memory are cheap, this is an excellent thing to do. The exponential step forward in software is tempting, but extremely risky.

Consider a simple task in a future workstation. My co-worker down the hall wants to know if I'm free for lunch. The corporate scheduling server reports that I don't have any meetings booked, so he sends me a short message. When I receive the message, it is beautifully typeset with anti-aliased fonts on my screen. A full color picture of the sender pops up as well. I reply, "Yes, let's eat at noon." Software automatically and beautifully formats the reply. An expert system disentangles the network routing. Perhaps 30 or 50 million instructions are executed to handle this trivial task, but those instructions are cheap and consume little time.

Connectivity

The biggest potential involves use of UNIX connectivity. High-speed ISDN networks will allow dramatic changes in the way we share information, world-wide.

Usenet and UUCP mail sprang up because communications tools existed to link computers anywhere, on demand, with ordinary dial-up telephones. The growth of Usenet was unplanned, and unpredicted. (Some may also say "undesirable", but that's another argument. For all the flaws, a new communications medium has arisen.)

With much more powerful networking software, and 100 times the dial-up bandwidth, what will we be able to do? If my computer can dial a machine on the other side of the continent, remotely mount a file system and communicate directly at high speed, many new opportunities should open up. But we will have to solve a number of implementation, security, and reliability problems. This will keep us busy, and burn more of those cheap MIPS and memory in the coming decade.

Performance

A particular challenge will be to match UNIX performance parameters and algorithms to the new high performance machines. We don't really know how to use that power. Since the power is cheap, some can be wasted, but it would be nice to really gain a factor of 100 improvement.

UNIX runs on supercomputers now, but it must be optimized for the interactive desktop supercomputer.

The Upper Layers — User and Applications Interfaces

The keynote address at this conference is entitled "Why We Have to Make UNIX Invisible." At this writing, I don't know anything about the contents of the talk, but I agree with the title at least.

UNIX will be the foundation for many applications solutions, but UNIX itself is *only* the foundation. Ordinary users won't be able to consume the available power by using today's user interfaces.⁶ Applications programs will need very high level interfaces. UNIX has the task of supporting such interfaces, and providing the glue that allows these interfaces to exist on all types and sizes of machine.

Standardization of UNIX means that certain low-level interfaces are hard to change. That's not a problem. What we need are new higher-level interfaces.

Personal Automation

The UNIX tool building approach together with convenient process creation has always meant a high degree of automation — reminders sent at night, *make*, daemons looking for work. UNIX has also allowed users to customize their environment to a larger extent than other systems. This has always been costly; processes are created "wastefully", process and command interpreter hierarchies are a dozen deep when re-making a system, etc. This expenditure of resources has bought flexibility.

The vast amounts of available storage and the large amount of available processing power opens up increased opportunities to create flexible UNIX systems. The challenge will be to control the complexity of this. We don't want to be swept away by a tide of shells, environment variables, editors, *ls* options, and so on. The complex interactions and the bugs can kill us, but many opportunities exist to harness the flexible power of our future systems.

When Does the World Really End?

In the computer industry we have enjoyed exponential progress for about 40 years. I expect at least another 15 years or more. But exponential curves continue forever only in mathematics.

Hardware performance/price ratios have improved by six orders of magnitude since the start of the computer industry. We certainly have a few more powers of 10 yet to come. But we will reach limits. Progress will not stop, but it will become linear or incremental.

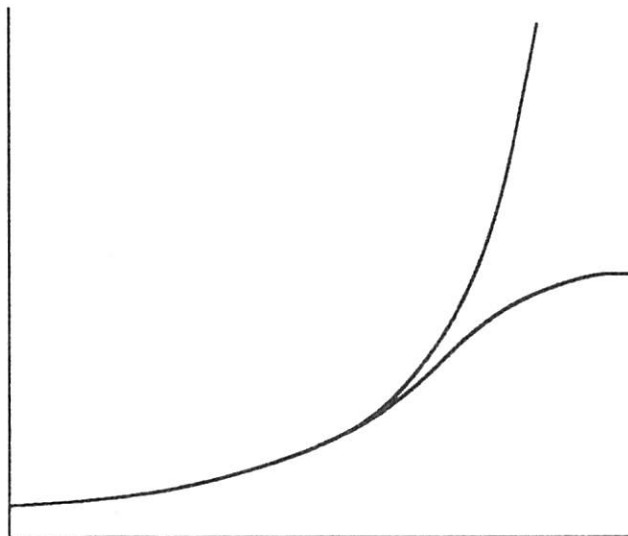
Although the year 1000 had no special significance for doomsday prophecies, the following century saw a number of such predictions. The year 2000 doesn't seem to be a barrier for progress in computing, but the chances appear good that we'll find some limits thereafter — a combination of quantum mechanics, speed-of-light, and software complexity.

⁶ Programmers, on the other hand, may have a smaller need for new interfaces. Since programmers work at an abstract level, they may have a greater need for better "power tools" rather than "user friendliness." The best programmers have an insatiable ability to consume raw resources, so improved hardware may be put to direct use.

Still, we in the software game (and in the UNIX subset of that game) can take heart. From the software point of view, progress has been much slower. We haven't really exploited all of the hardware power we have, and so there will be room for growth for a long time to come. If we do well, we will see UNIX as the basis for global computer connectivity. The longer development cycle for software means that we haven't yet fully exploited the UNIX system.

Eventually UNIX will be fully mature. The development of UNIX made life simpler. As machine power continues to increase and systems become more complex, a new simplifying step will become necessary.

This step hasn't yet been taken; it could be taken today but software inertia will delay the step until it's truly essential. If we do very well, we'll also see a replacement for UNIX on the horizon by the turn of the century.



Dream vs. Reality. Exponential curves eventually flatten in the real world. The upper curve is also known as the "hockey stick" curve. In start-up business plans, the blade of the stick occurs next year, regardless of year. In technology development, determining the inflection point is still a black art.

Acknowledgements

I would like to thank Brian Boyle of Novon Research Group for a wealth of data on machine cost and performance. Marsha Groves at the University of Toronto Centre for Medieval Studies contributed valuable historical perspectives. However, all technical or historical errors are solely my responsibility.

References

Brian Boyle, "The Mad Rush of Technological Progress," unpublished technical report, Novon Research Group, 1987.

Frederick P. Brooks Jr., "No Silver Bullet - Essence and Accidents of Software Engineering," *IEEE Computer*, April, 1987.

Norman Cohn, *The Pursuit of the Millennium*, rev. ed., 1970, Oxford University Press.

UTek Build Environment

Alan McIvor

Tektronix
Graphics Workstation Division

ABSTRACT

The software development environment created for the first releases of UTek (A Unix¹ like system based on 4.2 BSD²) was based on the assumption that there would be a single family of object code compatible products. That development environment supported a single hardware platform; adding any new platforms greatly increased the complexity of software development. The original system was not capable of supporting UTek on multiple platforms. Several problems arise when developing code for multiple platforms including software configuration control, cross development tools support, and machine dependencies maintenance. As a way to solve these problems the UTek Build Environment (UBE) provides a set of software tools needed to support the development of UTek on multiple hardware platforms.

1. Build Environment Issues

When designing the build environment we identified four key issues which we wanted to address with UBE. First, since we were supporting UTek on a variety of hardware platforms we needed a transparent way of maintaining machine dependencies. Our second goal was to identify a method to maintain the wide variety of software configurations and software packages available for UTek. The third task was to create a consistent way to deal with cross development tools. And lastly, the fourth task was to create an environment which wouldn't tie us to any particular hardware for the development environment.

2. Maintaining Machine Dependencies

While a majority of Unix is very generic (one of its strengths) there are a surprising number of machine dependencies. One task was to provide a way to identify and track these machine dependencies. We feel our source structure allows us to control the machine dependencies which exist within Unix. A related issue is maintaining compiled objects so programs for one processor do not work their way into a release for another processor.

¹ Unix is a Trademark of AT&T Bell Laboratories,

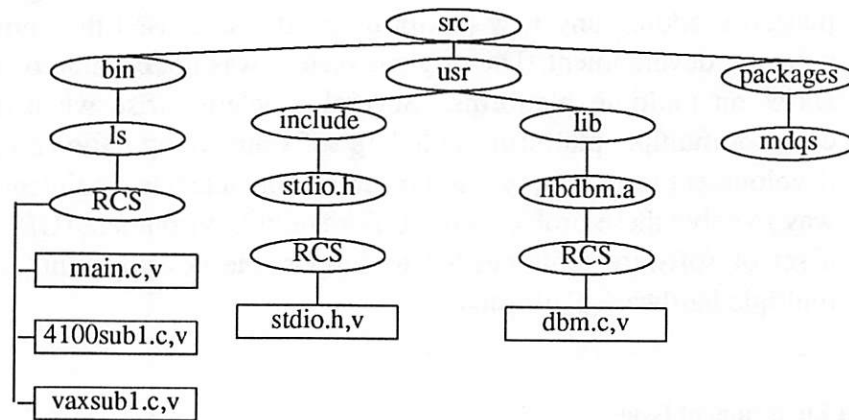
² BSD 4.2 is a Trademark of the Board of Regents of the University of California

2.1 Source Tree Organization

The source tree (\$ROOT/src, \$ROOT is a variable used by many utilities within UBE to define the root directory of UBE, many of the pathnames in this document use \$ROOT as part of the path) contains all the source code for UTEK. The goal in the design of the source tree structure was to provide a one-to-one correspondence between where a file is found within a UTEK release and its location within the source tree. The goal of one-to-one correspondence was not met 100%. There are certain utilities which need to be bundled together because they share include files and libraries (mdqs, uucp), these utilities are placed in a subtree of src called packages.

2.1.1 Standard Source

The basic format within the src tree is that for every utility or text file in UTEK there is a directory within the source tree; contained within that directory is an RCS directory. The RCS directory contains the source, makefile and any other supporting files necessary to build the utility. There are no directory entries in the source tree for files which are initially zero length. These files are created during the engineering release process.



2.1.2 Packages

The packages tree (\$ROOT/src/packages) contains utilities and libraries which share include files or libraries with other related utilities. Their directory structure is usually quite complex. We have tried to make the location of a utility in the packages subtree as close to its release location as possible. For example libc.a is in /src/packages/lib/libc.a. When it makes sense the utilities in the packages area are converted so they adhere more closely to the build environment structure. However, the packages tree will always remain as a standard way to do non-standard things within the build environment. For example, the packages part of the source tree is where we place master copies of binaries we receive from other organizations or third party vendors.

2.1.3 Multiple Platform Support

There are two ways to have multiple platform support: *ifdefs* within a section of code and separate modules. The build environment supports both methods. When doing a build for a particular machine *ifdefed* code is handled by defining certain values for the C preprocessor. If the source code for a module within a utility has a number of differences between platforms then the modules are maintained with the following naming convention: a prefix which indicates the platform for which that module is destined is added to the module name. For example if module_1.c for the Tek4100 series (NS32000 processor based workstation) version of some utility was drastically different from module_1.c for the VAX³ based product

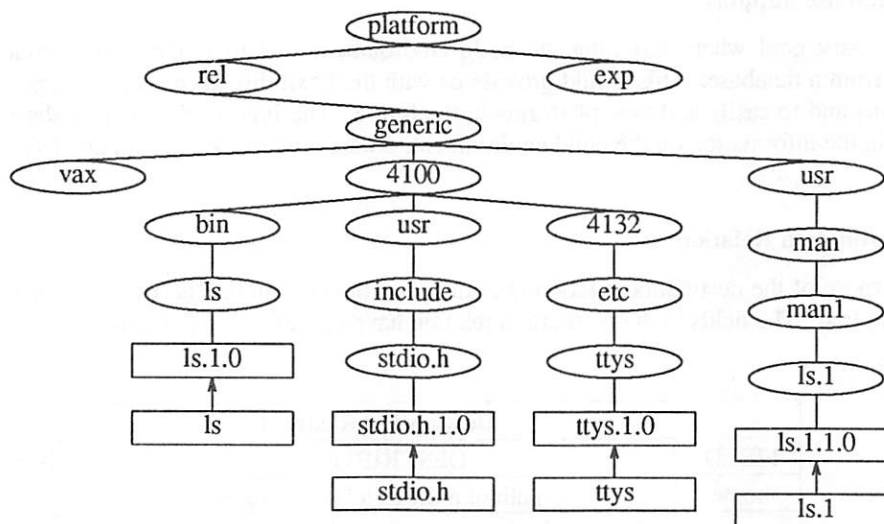
³ VAX is a registered trademark of Digital Equipment Corporation

the source files in the build environment would be 4100module_1.c and vaxmodule_1.c.

The build environment **make** and **makefiles** insure that the right module is used when building either a 4100 or VAX based utility. The same technique is also used for text files. If a text file is drastically different among the different platforms then there will be a text file for each platform with the platform name prepended to the text file name. There is a standard makefile template to insure utilities get built in a consistent fashion.

2.2 Platform Tree Organization

The platform tree is where all programs are installed after being built. It consists of two identical trees, one for production release (rel) builds and the other for experimental (exp) builds. Within the platform tree there are areas for storing files that are common to a particular machine, a family of machines or all machines.



2.2.1 Release and Experimental Trees

The rel branch of the platform tree (\$ROOT/platform/rel) is used when creating an engineering release. The exp branch (\$ROOT/platform/exp) holds the test builds for any utility, include file, or library.

2.2.2 Levels of Platform Tree

There are three levels to the platform tree: generic, family and machine specific. All files which are installed at the generic level may be considered common to all products. Examples of files at this level are manual pages and other text files common to all products. At the family level there are subtrees for every processor supported by UTek. The programs at the family level can be shared among all machines which are members of that family. Within each family there are subtrees for each of the machines which are members of that family. Placed in the machine subtree are files which are specific to that machine. Examples could be include files or system diagnostics.

2.2.3 Directory Format

The directory structure at each level of the platform tree is the same. To locate the include file /usr/include/sys/stat.h in the rel tree the following path would be searched:

\$ROOT/platform/rel/generic/\$FAMILY/\$MACHINE/usr/include/sys/stat.h.

The above path would be used assuming stat.h is machine specific. If stat.h were common to all machines for a given platform then this path would be used:

\$ROOT/platform/rel/generic/\$FAMILY/usr/include/sys/stat.h.

In the above example the product family name would be substituted for \$FAMILY and the machine name for \$MACHINE. When a file is installed into a directory in the platform tree the name has the following format, name.release.version. For example, the first time stat.h is installed it would become stat.h.1.0. Also created when a file is installed is a file which is hard linked to the most recently installed file within that directory. In the case of stat there would be a hard link stat.h which is linked to stat.h.1.0. The file without the release and version name (stat.h) is used by the build environment tools.

3. Software Configuration Control

3.1 Database Support

A secondary goal when designing the build environment was to provide an interface which would be driven from a database. This would provide us with the flexibility necessary to support multiple hardware platforms and to easily add new platforms in the future. The Ingres relational database package is used to maintain the information on the build environment. There are three key relations: destination, bom and history.

3.2 Destination Relation

The purpose of the destination relation is to maintain the information necessary to install software into the platform tree. The fields in the destination relation have the following definitions:

Table 1

Destination Relation		
FIELD	DESCRIPTION	Example
home	path of a file in a Utek release	/bin/ls
source	location in the source tree relative to \$ROOT/src	/bin/ls
class	level at which a file is installed in the platform tree	"f"
family	name of the platform family for this file	"4100"
machine	name of the machine for this file	"family"
release	release number for this build	1
version	version number for this build	0
rcsid	rcsid number for this file during this build	1.13

With the destination relation the following rules are followed for family and machine names. If a file is installed at the generic level then the family and machine fields are set to *generic*. If a file is installed at the family level then the family field is set to the name of the product family (4100, vax) and the machine field is set to *family*. Files which are installed at the machine level use the name of the machine (vax780, 4132) for the machine field and the family name for the family field. The destination relation also provides a mapping between the location of a file in the source tree and its location in a Utek release.

In the previous table the example column shows what type of information would be included in the destination relation. This example indicates that ls would be installed in /bin during a production release and the source for the utility can be found at \$ROOT/src/bin/ls. The class field "f" indicates the file is common to all products in the family ("g" is for generic files and "m" is for machine specific files). The family name is 4100. Since the utility is common to all machines in the family, the machine field is set to *family*. The release, version and rcsid fields indicate the most recent release of ls for the 4100 was 1.0 and that the rcsid number was 1.13. There is a tuple in the destination relation for every product family in which ls is included.

3.3 Bom Relation

Bill-Of-Materials (BOM) information is maintained by the bom relation. This relation is used to create the input files necessary to build the various packages. It is also used to create an engineering release of a given software package.

Table 2

BOM Relation		
FIELD	DESCRIPTION	EXAMPLE
home	path of a file in a UTek release	/bin/ls
owner	owner of this file in a UTek release	sys
group	group to which this file belongs in UTek release	sys
perm	permissions for this file	-r-xr-xr-x
size	size of this file	25600
link	link count for the file	6
package	BOM package to which this file belongs	4132core.bom
checksum	check sum	54635
linkto	file linked to, if it's a symbolic link	

The next table shows the attributes of the utility ls for the package 4100core.bom (base package for 4132 workstation). There would be a tuple in the bom relation for every package to which ls belongs. The size and checksum fields are set by the build environment tools when the utility is compiled and installed into the platform tree. If a change has to be made to an attribute of the file (ownership, permission) then the change only needs to be made here. The build environment tools set the ownership and permission of a utility when producing an engineering release.

3.4 History Relation

The history relation is used to maintain a relationship between a software release and the rcsid numbers of all the utilities that went into that release. This information allows us to go back and create a particular file from a given release or recreate a whole release.

Table 3

History Relation		
FIELD	DESCRIPTION	EXAMPLE
home	path of a file in a UTek release	/bin/ls
release	release number for this release of UTek	1
version	version within a major release	0
rcsid	rcsid number for this file	1.13
package	name of the package for this file	4132core.bom

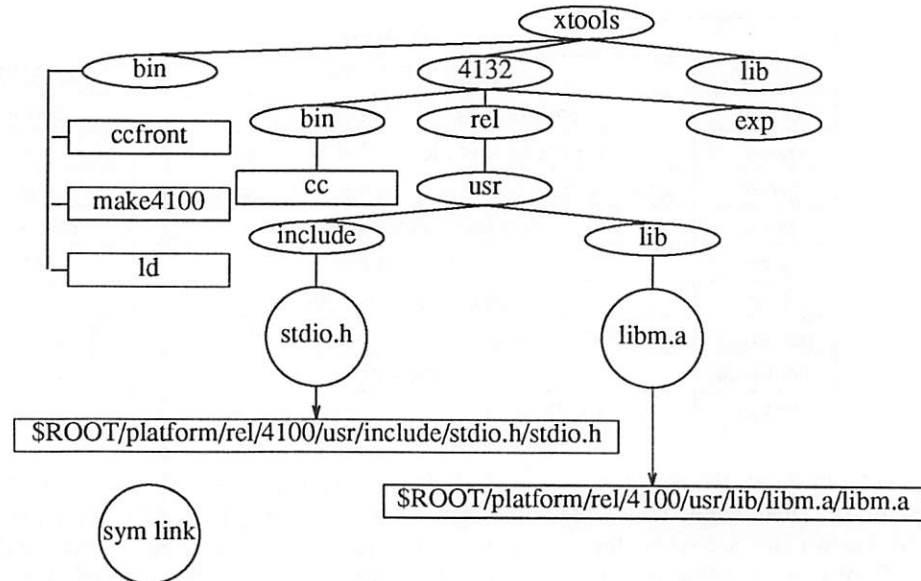
Every time a production release of ls is done a new tuple is added to this relation.

4. Tools Support

The cross tools tree (\$ROOT/xtools) contains everything which is needed to build a utility within the build environment. There are subtrees for tools which are common to all platforms (xtools/bin, xtools/lib) and subtrees for each machine supported by the build environment. The machine specific subtrees contain scripts, include file references and library references necessary to build a utility for a specific machine.

4.1 Machine Specific Trees

For each machine supported by the build environment there is a machine tree. The purpose of the machine tree is to provide a location for machine specific tools ($\$ROOT/xtools/\$MACHINE/bin$) and pointers to include files and libraries for both rel and exp builds.



4.1.1 Machine Bins

For each machine there is a bin directory which contains tools for that machine. These tools consist for the most part of front-end programs for **cc**, **ld**, etc. These tools create an environment in which the software can be compiled for a particular platform.

4.1.2 Machine Include Files and Libraries

Builds are done within the build environment by using the most recently installed version of include files and libraries. The product is built with the exact same include files and libraries as will be released with the product. Each include file or library in the xtools tree there is a symbolic link into the platform tree. This symbolic link points to the file hard linked to the most recently installed file in that directory.

$\$ROOT/xtools/4132/rel/lib/libc.a \rightarrow \$ROOT/platform/rel/generic/4100/lib/libc.a/libc.a$

Libc.a in the platform tree will point to the most recent version of libc.a which had been installed. Below the state directory (rel in the above example) the path to an include file or library in the xtools tree will be the same as that include file or library in a Utek release. The users of the build environment do not need to access the xtools tree. By using this cross tools configuration of a symlink pointing to a hard link the environment is guaranteed to use the most recent version of a include file or library. The xtools tree only has be created once for each machine and does not need to be updated each time an include file is updated.

4.2 Generic Tools

Contained in $\$ROOT/xtools/bin$ are commands which are used for building any particular platform. Examples are the generic C compiler front-end (**ccfront**) and the loader (**ld**). The generic programs are invoked by the machine specific front-end program. The front-end program sets up environment variables which are used by the generic tools to locate include files, libraries and other executables. Other programs placed in this directory are the build environment programs used for executing and controlling builds.

In \$ROOT/xtools/lib are the actual cross compilers for each processor as well as generic programs such as cpp.

5. Engineering Release Tree

The engineering release process consist of copying files from the rel branch of the platform tree to the engineering release tree (\$ROOT/eng_release). The structure of the eng_release tree is exactly the same as would be found on the destination platform with the base of the tree being \$ROOT/eng_release instead of "/". To build a release for a software package the **bom** and **destination** relations are used. For example, if the core package for the 4132 was to be built, the **bom** relation would be searched and the home field extracted for all tuples where the package field is equal to 4132core.bom. For each text file or utility found in the package 4132core.bom the **destination** relation is searched to find the path in the rel branch of the platform tree to that file or utility. The file is then copied from the platform tree to the eng_release tree. Lastly the file attributes (ownership, permission, etc) are set from the information contained in the **bom** relation.

Using the information from table 1 the path necessary to copy the utility ls from the platform tree to the eng_release tree can be constructed. First the tuple which contains the information for ls is found. This is done by examining the tuples whose source field is */bin/ls*. For each tuple whose source field is */bin/ls* the class field is checked to see if the utility is at the generic, family, or machine level of the platform tree. In table 1 the class field is set to *f*, so ls is installed at the family level. Next the family fields are searched to find the tuple for the product family for which this software build is being done; in this case the the family is *4100* (if the utility was being installed at the machine level instead of the family level then the machine field for each tuple would be compared against the machine name used for this build). All the information is now available to construct the path to ls in the platform tree. The path is:

\$(\$ROOT)/platform/rel/generic/4100/bin/ls/ls.

\$(\$ROOT)/platform/rel is used because releases are copied out of the rel branch of the platform tree. *Generic/4100* is part of the path because the file is installed at the family level. */bin/ls* is the path within the 4100 family to the directory which contains ls. The last component of the path *ls* is the hard link to the most recently installed version of ls.

6. Summary

There were four goals in the design of UBE. First was to maintain machine dependencies across platforms, second was to manage the wide variety of software configurations, third was to have a consistent cross tools structure and fourth was to have a system which would not tie Tektronix to a particular hardware development environment. The UTEK build environment solves several of the problems associated with supporting Unix in a multiple platform environment.

The structure of the source trees allows us to maintain machine dependencies in the source and easily identify the platform for which a module is destined. The platform tree prevents us from incorporating a utility or text file compiled or configured for one platform from working its way into a release for another platform.

Using the relational database Ingres, Tektronix can manage the large number of software packages we produce and create an engineering release of a package for a particular machine with a minimal number of commands. The database is also an integral part of our strategy of maintaining machine dependencies by providing a mapping of where a utility or text file is stored in the source tree and where it gets installed into the platform tree for a particular machine.

By integrating the cross tools tree and the platform tree Tektronix is able to insure the latest version of an include file or library is used during system builds. The tools for the UTEK Build Environment insure the right cross compiler, assembler, include files and libraries are used for any particular machine.

All of the UBE tools use the notion of a logical root (\$ROOT), therefor UBE can be relocated to any location on a machine or moved to another machine altogether. Through the use of UTEK's Distributed File System different parts of UBE can even reside on different machines.

Mk: a successor to make

Andrew Hume

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

research!andrew

ABSTRACT

Mk is an efficient general tool for describing and maintaining dependencies between files or programs. *Mk* is styled on, and largely compatible with the UNIX[®] tool *make*. The major advantages of *mk* over *make* are executing recipes in parallel, using pattern-matching metarules rather than suffix transformation rules, and deriving dependencies by transitive closure on all rules. *Mk* runs anywhere from 2 to 30 times faster than *make*.

This report describes *mk* by means of an evolving example. Other sections summarize the differences between *mk* and *make* and discuss the principles underlying *mk*'s design.

1. Introduction

A large fraction of computer activity consists of repeated application of tools (special or general purpose programs) to input files to produce output files. The most obvious example is programming, but other no less important examples range from simple document-processing pipelines to the generation of a circuit board or integrated circuit involving hundreds of files. Common to all these activities are file dependencies, where changing a file requires other files be remade. *Mk* reads a dependency description (called a *mkfile*) and does the minimal work necessary to bring a target file up to date.

Mk owes much to *make*, written by Stu Feldman, which has been doing a similar job on UNIX systems since 1976. The version of *make* referred to throughout this report is Feldman's research version distributed with Research Unix, Eighth Edition.

The next section is rather long. It follows the gradual development of a somewhat complicated *mkfile* describing how to build a C program. The third section summarizes the differences between *mk* and *make* and includes a comparison of execution times. The fourth section highlights the principles underlying *mk*. The appendix documents the predefined or builtin variables and rules for *mk*.

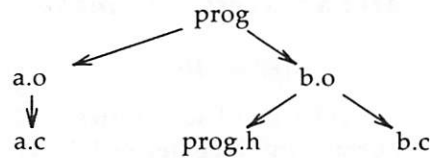
2. An Extended Example

This section describes *mk* in the context of building C programs. This is for the reader's comfort; *mk* knows nothing special about C programs. The example starts off small and simple and is extended throughout the section. Sometimes, *mk*'s behavior is best demonstrated by excerpts from a terminal session. These will be shown as

```
$ date
Fri Feb 20 20:06:03 EST 1987
$
```

where \$ is the prompt for the next command. Comments will be shown in *italics*.

Initially, our program is called `prog` and is made from `a.o` and `b.o`, which are made by compiling `a.c` and `b.c` respectively. In addition, `b.c` includes a header file `prog.h`. We represent these relationships pictorially below



The arrow means “depends on”. Thus, `prog` depends on `a.o` and `b.o` and if `a.o` or `b.o` is modified, then `prog` needs to be rebuilt. Similarly, `a.o` depends on `a.c` and `b.o` depends on `b.c` and `prog.h`.

The textual description of how `prog` is built is kept in a *mkfile* and looks like

```

prog: a.o b.o
      cc -o prog a.o b.o
a.o:  a.c
      cc -c a.c
b.o:  b.c prog.h
      cc -c b.c
  
```

The *mkfile* is a sequence of *rules*. Each rule defines a target (say `prog`) that depends on some prerequisites (`a.o` and `b.o`) and the commands (a shell script called the *recipe*) to bring the target up to date. *Mk* takes this description from a file named *mkfile* and builds the given targets. If no targets are given on the command line, the first target in the *mkfile* is built. For example, if we start with just the source files in our directory, *mk* creates `prog` by compiling `a.c` and `b.c`.

```

$ mk
cc -c a.c
cc -c b.c
cc -o prog a.o b.o
$
  
```

Executing *mk* again does nothing, as `prog` is now up to date.

```

$ mk
mk: `prog' is up to date
$
  
```

If we change a source file, *mk* rebuilds only the files that are out of date:

```

modify a.c
$ mk
cc -c a.c
cc -o prog a.o b.o
$
  
```

Mk will explain why it is rebuilding a file if we use the `-e` option. For example,

```

modify prog.h
$ mk -e
b.o(540869437) < prog.h(540869535)
cc -c b.c
prog(540869493) < b.o(540869546)
cc -o prog a.o b.o
$
  
```

Thus, `b.o` was out of date with respect to `prog.h`. After `b.o` was remade, `prog` was found

to be out of date with respect to `b.o` and was then rebuilt. The numbers are the actual time stamps of the files: the values are not as important as the difference between them. A time stamp of zero indicates a non-existent file.

Variables

Suppose we now need to compile the source files with the `-g` flag so that we can use the debugger. We can of course simply edit each rule to change `cc` into `cc -g`:

```
prog: a.o b.o
      cc -g -o prog a.o b.o
a.o:  a.c
      cc -g -c a.c
b.o:  b.c prog.h
      cc -g -c b.c
```

A better solution is to use a *variable*. A *mk* variable has a similar form and use to a shell variable. A suitable (mnemonic) name is `CFLAGS`. The new *mkfile* looks like this:

```
CFLAGS=-g
prog: a.o b.o
      cc $CFLAGS -o prog a.o b.o
a.o:  a.c
      cc $CFLAGS -c a.c
b.o:  b.c prog.h
      cc $CFLAGS -c b.c
```

Now, if we want to profile `prog` (which means compiling everything with the `-p` option), we need only change the first line to

```
CFLAGS=-g -p
```

and recompile all the object files. The easiest way to recompile everything is with `mk -a` which says to always make every target regardless of time stamps.

Some variables are supplied by *mk* for use by the recipe. One is `prereq` whose value is all the prerequisites for this rule. We can rewrite the first rule like this:

```
prog: a.o b.o
      cc $CFLAGS -o prog $prereq
```

This guarantees that the lists of object files (the prerequisite line and the `cc` line) are the same. It is now easy to incorporate a new object file `c.o` by adding the new name just once:

```
CFLAGS=-g -p
prog: a.o b.o c.o
      cc $CFLAGS -o prog $prereq
a.o:  a.c
      cc $CFLAGS -c a.c
b.o:  b.c prog.h
      cc $CFLAGS -c b.c
c.o:  c.c prog.h
      cc $CFLAGS -c c.c
```

Metarules

The preceding rules for the `.o` files are very similar. *Mk* supports *metarules*, that is, rules that apply to a class of targets, rather than just one specific target. The class of targets is defined by pattern matching, with the symbol `%` (called the stem) equivalent to the regular expression `.*`. For example, the normal rule for compiling C source files is

```
%.o: %.c
      $CC $CFLAGS -c $stem.c
```

The variable `stem` in the recipe is the string matched by the `%`. The `CC` variable is good planning; a different compiler can be used very easily. Using this metarule, our `mkfile` becomes much shorter:

```
CC=cc
CFLAGS=-g -p
prog: a.o b.o c.o
      $CC $CFLAGS -o prog $prereq
b.o:  prog.h
c.o:  prog.h
%.o:  %.c
      $CC $CFLAGS -c $stem.c
```

Notice that the prerequisites for a target can be spread across many rules. Two rules apply to `b.o`, the specific rule with `prog.h` and the metarule for `.o`'s. Only one of the rules should have a recipe. If there is more than one recipe, *mk* complains that the way to make the target is ambiguous.

The `%` can appear anywhere in the target or prerequisite, not just at the beginning.

Mk has some predefined variables and rules listed in Appendix 1. Because our rule for `%.o` and the value for `CC` are the same as the predefined rules and variables, we can omit them for a shorter `mkfile`:

```
CFLAGS=-g -p
prog: a.o b.o c.o
      $CC $CFLAGS -o prog $prereq
b.o:  prog.h
c.o:  prog.h
```

Any non-metarule takes precedence over a metarule. Thus, metarules for generating `.o`'s (say) do not conflict with any rule for generating a specific `.o`.

Rules with no prerequisites

Rules need not actually build their targets. Some rules are simply shell scripts embedded in the `mkfile` for convenience. For example, most `mkfiles` have the target `clean`:

```
clean:
      rm -f *.o prog core
```

Note that `clean` is intended as a label, not a file. Unfortunately, if a file named `clean` exists, the recipe will not be executed, since `clean` is up to date (because no prerequisite has caused it to be out of date). We want to avoid any such inadvertent interactions with the file system. *Mk* allows a label to have an attribute of *virtual*, which means that it is distinct from a file of the same name. Targets can be marked as virtual by appending a `V:` to the colon separator between targets and prerequisites:

```
clean:V:
      rm -f *.o prog core
```

Other attributes are described below.

Rules with multiple targets

The rules relating `b.o` and `c.o` to `prog.h` can be combined into one rule with two targets.

```

CFLAGS=-g -p
prog: a.o b.o c.o
      $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
clean:V:
      rm -f *.o prog core

```

If a rule with multiple targets has no recipe, it is simply a shorthand notation for all the simple rules with one target. A rule with multiple targets and a recipe has subtle implications described below. To motivate the subtleties, we digress to describe the *yacc* parser generator.

Yacc takes a file describing a grammar and produces the source for a C routine that will parse input according to the given grammar. The source is put in the file `y.tab.c`. *Yacc* also produces a header file called `y.tab.h` that links the parser to a lexical analyzer. The grammar file also contains semantic action code. Typically, changes to the grammar file do not change the header `y.tab.h`, but only the semantic routines.

Let us add a grammar and a lexical analyzer to `prog*`:

```

prog: a.o b.o c.o y.tab.o lex.o
      $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: y.tab.h
y.tab.c y.tab.h:      gram.y
                  yacc -d gram.y

```

The grammar is kept in `gram.y` (the conventional suffix for *yacc* input is `.y`). The `-d` option to *yacc* produces `y.tab.h`. Unfortunately, this mkfile does too much work in the normal case. Every time the grammar file is changed, a new `y.tab.h` is made and thus, `lex.o` will always be out of date even though the contents of `y.tab.h` may not have been changed. The best solution maintains another header file (say `x.tab.h`) that only changes when necessary, that is, when the contents of `y.tab.h` actually change. The new mkfile is

```

prog: a.o b.o c.o y.tab.o lex.o
      $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: x.tab.h
x.tab.h:      y.tab.h
              cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:      gram.y
                  yacc -d gram.y

```

The recipe for `x.tab.h` is a conditional shell construct; if the command `cmp -s x.tab.h y.tab.h` returns with an error (the files are different), then execute the command `cp y.tab.h x.tab.h` to copy `y.tab.h` onto `x.tab.h`. In the case where `y.tab.h` doesn't change, the action is straightforward:

*Some unimportant detail has been removed from the mkfile.


```

$ mk -e
y.tab.c(541051073) < gram.y(541051092)
y.tab.h(541051072) < gram.y(541051092)
yacc -d gram.y
y.tab.o(541051082) < y.tab.c(541051100)
cc -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051099)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp not done
prog(541051087) < y.tab.o(541051109)
cc -o prog a.o b.o c.o y.tab.o lex.o
$

```

If we now change the grammar so that the header file does change:

```

$ mk -e
y.tab.c(541051100) < gram.y(541051148)
y.tab.h(541051099) < gram.y(541051148)
yacc -d gram.y
y.tab.o(541051109) < y.tab.c(541051155)
cc -c y.tab.c
x.tab.h(541042236) < y.tab.h(541051154)
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cp done; x.tab.h updated
lex.o(541042267) < x.tab.h(541051165)
cc -c lex.c
prog(541051114) < y.tab.o(541051163)
prog(541051114) < lex.o(541051169)
cc -o prog a.o b.o c.o y.tab.o lex.o
$

```

The subtleties are twofold. The first is that the time stamps for files are only examined when the file is initially referenced or when it is the target of a rule. If `y.tab.h` had not been a target for the `yacc` rule, then `mk` would assume that `y.tab.h` had not been updated. The second subtlety is that the rule for `x.tab.h` need not change `x.tab.h`. If it does not, then `lex.o` need not be recompiled.

Aggregates

Some of the things we would like to maintain with `mk` are actually collections or *aggregates* of entities, such as Unix object libraries (archives maintained by `ar`). Other (unsupported as yet) examples are `cpio` and SCCS files. The type of aggregate is determined by the file's "magic number". Each type has support code within `mk` to get the time stamp of a member and to "touch" (see below) a member. The notation `a(m)` refers to member `m` of aggregate `a`. For example, consider an archive `lib.a` made up of `a.o`, `b.o`, and `c.o`. The `mkfile` looks like

```

lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
lib.a(%.o): %.o
        ar r lib.a $stem.o

```

As each new `.o` file is generated, it is put into `lib.a`. This is straightforward and correct but inefficient. An `ar` command is executed for every out of date object file. A better way is to generate all the `.o` files and then do the `ar`. The new `mkfile` relies on a shell script called *membername*:

```
lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
       ar r lib.a `membername $newprereq`
lib.a(%o): %o
```

Membername takes aggregate notation and extracts the member names. For example,

```
$ membername 'lib.a(a.o)' 'lib.a(b.o)' 'lib.a(c.o)'
a.o b.o c.o
$
```

The quotes are to stop the shell from interpreting the (). We use the variable *newprereq* because we only need to replace the object files that have changed.

Parallel processing

Mk executes recipes by continually traversing the dependency graph looking for targets that can be made. For example, in our *mkfile*:

```
prog: a.o b.o c.o y.tab.o lex.o
      $CC $CFLAGS -o prog $prereq
b.o c.o:      prog.h
lex.o: x.tab.h
x.tab.h:      y.tab.h
           cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
y.tab.c y.tab.h:      gram.y
           yacc -d gram.y
```

the target *a.o* can be made immediately, while the target *y.tab.o* has to wait for *y.tab.c* to be made. When *mk* finds a recipe it can execute, it puts the recipe on a queue. When the recipe terminates, *mk* updates the dependency graph. The number of recipes executing simultaneously is the value of the variable *NPROC*, which is initially one. On multi-processor machines, *mk* goes faster with higher values; most *mkfiles* on our 12 processor machine have *NPROC* between 6 and 10. In most situations, increasing *NPROC* beyond a certain limit gains almost nothing. The other way to speed up parallel builds is to ensure that as many recipes as possible are executing; that is, order the sub-targets such that the slowest are done first. While *mk* gives no guarantees about the order of builds, generally prerequisites are built in left-to-right order as in the *mkfile*. The *-u* (utilization) option measures how many seconds (real time) are spent with so many recipes executing. For example, building *prog* with three simultaneous recipes yields

```
0: 1
1: 4
2: 7
3: 10
```

The time with zero recipes executing corresponds to *mk* reading the *mkfile* and building the dependency graph.

Parallel execution implies that recipes should not interact unnecessarily. For example, the first version of the library *mkfile* should not be run in parallel as simultaneous *ar*'s on the same archive interfere*. The second version can because only one *ar* is done: after all the object files are made.

*Arguably, *mk* might protect against simultaneous updates of an aggregate but that is currently infeasible because it implies understanding what the recipe does.

Missing intermediates

In all the examples we have seen so far, *mk* has made all the targets “between” the file that changed and the main target. This is not always done. Any non-existent intermediate target (a target other than the root target with prerequisites) is treated specially. If pretending it existed with the time stamp of its most recent prerequisite would make all targets that depended on it be up to date, then it is not made. For example, in our mkfile:

```
$ mk -e
mk: 'prog' is up to date
remove a.o
$ mk -e
pretending a.o has time 540869454
mk: 'prog' is up to date
```

The intuition is that if we use the mkfile to build the targets, then removing the intermediates causes no harm. Of course, if we actually need the missing intermediates, *mk* builds them.

```
change b.c
$ mk -e
pretending a.o has time 540869454
b.o(540869546) < b.c(541350226)
cc -c b.c
unpretending a.o because of prog because of b.o
a.o(0) < a.c(540869454)
cc -c a.c
prog(541104056) < a.o(541350255)
prog(541104056) < b.o(541350244)
cc -o prog a.o b.o c.o y.tab.o lex.o
$
```

The action is not too hard to follow: first *mk* sees that *a.o* is missing and pretends it is there. Then *mk* notices *b.o* is out of date and needs to be rebuilt. When *b.o* is finally built, it causes *prog* to become out of date and therefore *mk* no longer can pretend that *a.o* is up to date. It then builds *a.o* and then *prog*.

The major advantage of missing intermediates is avoiding multiple copies of files. For example, in our mkfile to maintain a library, we keep two copies of every object file. By using the notion of missing intermediates, we can keep one copy — the copy we need in the archive. To do so, simply remove the object files after they have been archived:

```
lib.a: lib.a(a.o) lib.a(b.o) lib.a(c.o)
      names='membername $newprereq'
      ar r lib.a $names && rm $names
lib.a(%.o): %.o
```

We store the object files' names in the variable *names* to avoid executing *membername* twice. The *&&* is another conditional shell construct; we remove the files only if the archive command succeeds.

The special treatment of missing intermediates is suppressed by the *-i* option of *mk*.

Administrative

Mk provides an easy way to bring a target up to date without actually doing any work. For example, if we change *prog.h* in such a way that *b.o* or *c.o* won't change (such as adding a comment), we don't want to recompile the files. Instead, we can ask *mk* to modify the files' time stamps.

```

add something to prog.h
$ mk -t
touch(b.o)
touch(c.o)
touch(prog)
$

```

Mk lists the files it modified. This is a dangerous feature; use it carefully and sparingly. Virtual targets are not affected because *touching* only changes files.

Mk can also tell us what it would do without actually doing it. The option *-n* causes recipes to be printed rather than executed. There are two main problems. *Mk* assumes that every recipe will update all its targets. Normally this is true, but for our mkfile, *mk -n* would erroneously indicate that *lex.o* will always be remade. Thus, unnecessary work may be indicated. The second problem is that *mk* expands recognizable references to shell variables. It does this without parsing the shell script and can make mistakes with constructs like *for* loops. For example, with the mkfile (the *Q* attribute suppresses the normal recipe echo)

```

i=a b c
all:Q:
    for i in x y z
    do
        echo $i
    done

```

the difference between *mk* and *mk -n* is:

```

$ mk -n
for i in x y z
do
    echo a b c
done
$ mk
x
y
z
$

```

This latter problem applies to the normal recipe echo as well.

Sometimes we would like to know what *mk* would do if some files were changed. The *-wfiles,...* option supports this "what if" query by setting the time stamps internally for the named files to the current time. With our mkfile for *prog*, we can ask what would happen if we changed *prog.h*:

```

$ mk -n -wprog.h
cc -c b.c
cc -c c.c
cc -o prog a.o b.o c.o y.tab.o lex.o
$

```

The advantage of *-w* is that neither the files nor their time stamps are changed.

More on metarules

There are actually two kinds of metarules; we have looked only at the kind that uses *%* to match arbitrary strings. The second kind uses full regular expressions as supported in *egrep*(1). The expression may include sub-expressions enclosed in *\(\)*; the values of the sub-expressions can be used in the prerequisites and the recipe. For example, consider the

problem of making object files in sub-directories. That is, we wish to make `dir/a.o` from `dir/a.c`. The C compiler only generates object files in the current directory, so we need to break the target into two parts:

```
\(.*\)/\[^\/*\]\.o:R:    \1/\2.c
    cd $stem1; $CC $CFLAGS -c $stem2.c
```

The `R` attribute for the rule means interpret the target(s) as regular expression(s). The different ways of referring to the sub-expressions are regrettable. The `\\` are necessary within the prerequisites because of the shell quoting rules. A warning: regular expression metarules are significantly slower than `%` metarules.

Regardless of which kind of metarule you use, certain metarules can lead to infinite dependency graphs. For example, the metarule

```
%.z:    %.z
    unpack $stem.z
```

gives this dependency graph

```
x      →    x.z      →    x.z.z      →    x.z.z.z      →    ...
```

The problem arises any time a metarule has a prerequisite that can be a target of the same rule. *Mk* handles this problem by restricting the number of times a metarule is used in generating prerequisites to the value of the variable `NREP`. This value is normally one; if set to 3, the dependency graph for `x` in our example is

```
x      →    x.z      →    x.z.z      →    x.z.z.z
```

Thus, setting `NREP` to greater than one is necessary if we have files that have been packed repeatedly.

3. Differences between *make* and *mk*

The qualitative differences between *mk* and *make* can be summarized as

- *Make* builds targets when it needs them, allowing systematic use of side effects. *Mk* constructs the entire dependency graph before building any target.
- *Make* supports suffix rules and `%` metarules. *Mk* supports `%` and regular expression metarules.
- *Mk* performs transitive closure on metarules, *make* does not.
- *Make* supports cyclic dependencies, *mk* does not.
- *Make*'s recipes are collections of one-line shell commands, executed a line at a time. Variable values are passed by editing the recipe text before passing it through to the shell. *Mk*'s are simply shell scripts executed as one unit. Variable values are passed through environment variables.
- *Make* supports parallel execution of single line recipes when building the prerequisites for specified targets. *Mk* supports parallel execution of all recipes.
- *Make* uses special targets (beginning with a `.`) to indicate special processing. *Mk* uses attributes indicated by qualifiers after the `:` separator in a rule definition.
- *Mk* allows the standard output of a recipe to be read as an additional mkfile while *mk* is running. This allows a mkfile to configure itself at run time.

In most situations, mkfiles and makefiles (the input for *make*) will have only minor syntactic differences. In practice, mkfiles often are significantly bigger because of embedded shell scripts or to make the most of underlying parallel hardware.

The most striking difference between *mk* and *make* is in speed of execution. There are three main factors involved. *Make* uses a linear list to access variables and rules; *mk* uses a hash table. *Mk* and *make* use time stamps in slightly different ways; *make* often has to

measure a file's time stamp unnecessarily. If there are metarules, *mk* will typically create a much larger dependency graph than *make*. The graph gets pruned but at the cost of testing (for existence) a large number of files. In the examples given below, execution times are given (in seconds) as a sum of user time (a measure of how efficiently the dependency graph is built and executed) and system time (a measure of how many time stamps are measured). The times do not include times for recipe executions.

For mkfiles with no metarules, *mk* is always faster than *make* because of better accessing algorithms. For example, the mkfile to compile the operating system describes 83 object files. *Make* takes 19.8u+3.6s, *mk* takes 6.6u+3.6s. *Mk* is faster by a factor of 3 (user time) and 2.3 (user+sys).

For more normal mkfiles (that use the builtin metarules), *make* is somewhat faster than *mk* until about a dozen prerequisites are involved. *Mk* is much better for larger mkfiles. In most cases, *mk*'s performance can be improved by only using necessary metarules. For example, for a program made from 61 object files all compiled from .c files, we give the times for a normal mkfile and a mkfile that has only one metarule (generating %.o from %.c).

Command	Run Time	Relative Speed (user)	Relative Speed (user+sys)
make	12.0u+9.7s	1	1
mk (all metarules)	5.1u+4.0s	2.3	2.4
mk (one metarule)	3.9u+2.9s	3	3.2

Mk handles aggregates efficiently. The main C library has 242 members. *Make* takes 47.7u+10.9s, *mk* takes 6.3u+12.5s. *Mk* is faster by a factor of 7.6 (user time) and 3.1 (user+sys).

The final example comes from Ted Kowalski at AT&T Bell Laboratories. The mkfile is about 20,000 characters and describes an experimental workstation environment built from 238 .c files, 59 .h files, 7 .y files and 7 .l files. The mkfile makes heavy use of variables. *Make* takes 278.8u+16.2s, *mk* takes 8.4u+10.5s. *Mk* is faster by a factor of 33 (user time) and 15.6 (user+sys).

Despite the marked speed advantage of *mk* over *make*, the main reason users in our computing community use *mk* is its functionality, in particular, transitive closure on metarules, parallel execution of recipes, and the regular expression metarules.

4. The Principles

Mk's semantics and syntax were designed according to a few general principles or guidelines.

Use existing syntax and notions. The syntax of mkfiles is almost exactly the same as a makefile (used by *make*). (The only syntactic change for rules is the attribute marking.) *Mk*'s variables are exactly the same as shell variables. Recipes are written in *sh*(1), not a special purpose language. The regular expression syntax and semantics were adopted from existing tools (such as *egrep* and *ed*), trading some awkwardness for familiarity.

Generalize features. *Make*'s metarules (already a generalization of the early *make* sufix rules) were extended to full regular expressions. *Mk* performs the transitive closure on the target-prerequisite relations defined by all rules, including metarules. The primitive form of parallel processing supported by *make* has been generalized to allow parallel execution of any recipe. By constructing the entire dependency graph before executing any recipes, *mk* maximizes the benefits from parallel processing.

Removing special cases. *Make*'s variables and recipes were so close to being shell variables and scripts that the differences were removed in *mk*. Making recipes shell scripts had the further advantage that *mk* does not have to parse or process the recipes. The use of special target and prerequisite names (beginning with a dot) to indicate special actions has been dropped in favor of a more explicit notion of target attributes.

Mk is a general purpose tool. Recent versions of *make* (such as *nmake*) focus on the issues connected with building software and generally contain much builtin knowledge about C programming. *Mk*, on the other hand, is a tool for maintaining file dependencies, whether they be programs or circuit board descriptions. It offers general purpose and powerful mechanism for all users, not just help for programmers.

5. Appendix

The following variable definitions are made before processing the environment or any mkfiles.

```
AS=as
CC=cc
CFLAGS=
FC=f77
FFLAGS=
LDFLAGS=
LEX=lex
LFLAGS=
NPROC=1
NREP=1
YACC=yacc
YFLAGS=
```

The builtin rules are

```
%.o: %.c
    $CC $CFLAGS -c $stem.c
%.o: %.s
    $AS -o $stem.o $stem.s
%.o: %.f
    $FC $FFLAGS -c $stem.c
%.o: %.y
    $YACC $YFLAGS -o $stem.c $stem.y &&
    $CC $CFLAGS -c $stem.c && rm $stem.c
%.o: %.l
    $LEX $LFLAGS -t $stem.l > $stem.c &&
    $CC $CFLAGS -c $stem.c && rm $stem.c
```

The environment for the recipe's shell is augmented by these variables:

alltarget	all the targets for this rule.
newprereq	the prerequisites that are more recent than the target.
nproc	this is the process slot for this recipe. It is a number between zero and \$NPROC-1 inclusive. It is useful for parallel execution on a single CPU machine on a network.
pid	the process id for the <i>mk</i> invoking this script. This is useful for communicating with other rules.
prereq	all the prerequisites for this target. This may include prerequisites from several rules.
stem,...	the value of % in a metarule. It is null for a non-metarule. The value of the <i>n</i> th subexpression in a regular expression metarule is put in the variable <i>stem_n</i> , for <i>n</i> <10. It is null otherwise.
target	the targets being built for this rule.

Miranda — An Advanced Functional Programming System Running Under UNIX

David Turner
Computing Laboratory
University of Kent
Canterbury CT2 7NF
ENGLAND
mcvax!uke!dat

There has been an upsurge of interest in functional programming in the in the last few years, and a great deal of progress has been made in the design of functional languages. Two discoveries in particular have transformed the state of the art. These are — lazy evaluation — and polymorphic strong typing.

Lazy evaluation (the principle that no subexpression should be evaluated until its value is known to be required) makes it possible to write functional programs that describe and use infinite data structures. This opens up some interesting programming techniques not available in conventional languages. It also allows us to handle problems of input/output and communicating processes in a purely functional style, whereas earlier functional languages (such as LISP) had to cheat by introducing side effects to handle problems of this kind.

Polymorphic strong typing (originally developed by Milner for ML) is an equally important development. It combines the standard advantage of strong typing (that all type errors are detected at compile time) with the ability to define generic functions (such as 'reverse', 'sort' etc.), that can be used at many different types. A type system of this kind is based on type inference (by the compiler) rather than on type declaration (by the user) so one can write nice succinct programs, uncluttered by type information, and yet enjoy the full security of a compile time type discipline.

These two major developments (and a number of minor notational improvements) have transformed the state of functional programming very substantially from what it was say ten years ago. The Miranda system is an attempt to make these programming techniques available to a larger community, by providing a stable and well documented implementation of a modern functional programming language, embedded in a high quality programming environment running under the UNIX operating system (*). The purpose of this paper is to give a brief overview of the main features of Miranda.

Miranda is a simple, strongly typed, functional programming language based on higher order recursion equations. It draws its main features from the earlier languages SASL, KRC and ML, but with significant innovations in the area of user defined types. The Miranda system is a product of Research Software Limited, and runs on a variety of computers under UNIX (brief information about availability is given at the end of this paper).

(*) UNIX is a trademark of A T & T. Bell Laboratories. Miranda is a trademark of Research Software Limited.

The programming environment

The Miranda system is interactive and runs under UNIX as a self contained subsystem. A Miranda program is a collection of definitions, called a 'script'. Each script is stored in a UNIX file, and at any given time there is a 'current script', identified by a UNIX pathname. The basic action of the Miranda system is to evaluate expressions, supplied by the user at the terminal, in the environment established by the current script. So in its basic mode of operation the system is somewhat like a desk calculator, but with the ability to invoke user defined functions and data structures.

The Miranda compiler works in conjunction with an editor (normally this is 'vi' but it can be set to any editor of the user's choice). Scripts are automatically recompiled after edits, and any syntax or type errors signalled immediately. If the script does contain a compile-time error, the editor automatically reopens the file with the cursor positioned at the line containing the error (first error if more than one). Because of the type system a high proportion of programming errors are detected at compile time.

There is quite a large library of standard functions. There is also an online reference manual. There is reasonably powerful interface to UNIX, permitting Miranda functions to obtain data from, and send results to, arbitrary UNIX files, and it is possible to install a Miranda function as a stand-alone UNIX command, so it can be invoked directly from the UNIX shell.

The Miranda system has a number of other useful features in its user interface (which we skip here for lack of space) which help to make it a high-productivity environment for the development of functional programs. Because this environment is to a large extent constructed from standard UNIX components it is relatively portable between UNIX hosts and the code of the Miranda system proper is small (under 300K for the whole system including the compiler) in relation to the large amount of functionality which it provides.

The remainder of this paper is taken up with a discussion of the Miranda functional programming language (i.e. the notation in which Miranda scripts are written).

Basic ideas

The Miranda programming language is purely functional — there are no side effects or imperative features of any kind. A program (as remarked we don't call it a program, we call it a 'script') is a collection of equations defining various functions and data structures which we are interested in computing. The order in which the equations are given is not in general significant. There is for example no obligation for the definition of an entity to precede its first use. Here is a very simple example of a Miranda script:

```
z = sq x / sq y
sq n = n * n
x = a + b
y = a - b
a = 10
b = 5
```

If the above is our current script and we type 'z' into the system, the result will be to print '9' on the screen.

Notice the absence of syntactic baggage — Miranda is, by design, rather terse. There are no mandatory type declarations, although (see later) the language is strongly typed. There are no semicolons at the end of definitions — the parsing algorithm makes intelligent use of layout. Note that the notation for function application is simply juxtaposition, as in 'sq x'. In the definition of the sq function, 'n' is a formal parameter — its scope is limited to the equation in which it occurs (whereas the other names introduced above have the whole script for their scope).

The most commonly used data structure is the list, which in Miranda is written with square brackets and commas, eg:

```
week_days = ["Mon","Tue","Wed","Thur","Fri"]
days = week_days ++ ["Sat","Sun"]
```

Lists may be appended by the “++” operator. Other useful operations on lists include infix “:” which conses an element at the front of a list, “#” which takes the length of a list, and infix “!” which does subscripting. So for example 0:[1,2,3] has the value [0,1,2,3], #days is 7, and days!0 is “Mon”.

There is also an operator “--” which does list subtraction. For example [1,2,3,4,5] -- [2,4] is [1,3,5].

There is a shorthand notation using “..” for lists whose elements form an arithmetic series. Here for example are definitions of the factorial function, and of a number “result” which is the sum of the squares of the odd numbers between 1 and 100 (sum and product are library functions):

```
fac n = product [1..n]
result = sum[1,3..100]
```

The elements of a list must all be of the same type. A sequence of elements of mixed type is called a tuple, and is written using parentheses instead of square brackets. Example

```
employee = ("Jones",True,False,39)
```

Tuples are analogous to records in Pascal (whereas lists are analogous to arrays). Tuples cannot be subscripted — their elements are extracted by pattern matching (see later).

Guarded equations and block structure

An equation can have several alternative right hand sides distinguished by “guards” (a guard is a boolean expression written following a comma). So for example the greatest common divisor function can be written:

```
gcd a b = gcd (a-b) b, a>b
        = gcd a (b-a), a<b
        = a, a=b
```

It is also permitted to introduce local definitions on the right hand side of a definition, by means of a “where” clause. Consider for example the following definition of a function for solving quadratic equations (it either fails or returns a list of one or two real roots):

```
quadsolve a b c = error "complex roots", delta<0
                = [-b/(2*a)], delta=0
                = [-b/(2*a) + radix/(2*a),
                  -b/(2*a) - radix/(2*a)], delta>0
    where
    delta = b*b - 4*a*c
    radix = sqrt delta
```

Where clauses may occur nested, to arbitrary depth, allowing Miranda programs to be organised with a nested block structure. Indentation of inner blocks is compulsory, as layout information is used by the parser.

Pattern matching

It is permitted to define a function by giving several alternative equations, distinguished by the use of different patterns in the formal parameters. This provides another method of doing

case analysis which is often more elegant than the use of guards. We here give some simple examples of pattern matching on natural numbers, lists, and tuples. Here is (another) definition of the factorial function, and a definition of ackerman's function:

```
fac 0 = 1
fac (n+1) = (n+1)*fac n

ack 0 n = n+1
ack (m+1) 0 = ack m 1
ack (m+1) (n+1) = ack m(ack (m+1) n)
```

Here is a (naive) definition of a function for computing the n'th fibonacci number:

```
fib 0 = 0
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

Here are some simple examples of functions defined by pattern matching on lists:

```
sum [] = 0
sum (a:x) = a + sum x

product [] = 0
product (a:x) = a * product x

reverse [] = []
reverse (a:x) = reverse x ++ [a]
```

Accessing the elements of a tuple is also done by pattern matching. For example the selection functions on 2-tuples can be defined thus

```
fst (a,b) = a
snd (a,b) = b
```

As final examples we give the definitions of two Miranda library functions, take and drop, which return the first n members of a list, and the rest of the list without the first n members, respectively

```
take 0 x = []
take (n+1) [] = []
take (n+1) (a:x) = a : take n x

drop 0 x = x
drop (n+1) [] = []
drop (n+1) (a:x) = drop n x
```

Notice that the two functions are defined in such a way that the following identity always holds — “take n x ++ drop n x = x” — including in the pathological case that the length of x is less than n.

Currying and higher order functions

Miranda is a fully higher order language — functions are first class citizens and can be both passed as parameters and returned as results. Function application is left associative, so when we write “f x y” it is parsed as “(f x) y”, meaning that the result of applying f to x is a function, which is then applied to y. The reader may test out his understanding of higher order functions by working out what is the value of “answer” in the following script:

```

answer = twice twice twice suc 0
twice f x = f(f x)
suc x = x + 1

```

Note that in Miranda every function of two or more arguments is actually a higher order function. This is very useful as it permits partial parameterisation. For example “member” is a library function such that “member x a” tests if the list x contains the element a (returning True or False as appropriate). By partially parameterising member we can derive many useful predicates, such as

```

vowel = member ['a','e','i','o','u']
digit = member ['0','1','2','3','4','5','6','7','8','9']
month = member ["Jan","Feb","Mar","Apr","Jun","Jul","Aug","Sep",
               "Oct","Nov","Dec"]

```

As another example of higher order programming consider the function foldr, defined by

```

foldr op k [] = k
foldr op k (a:x) = op a (foldr op k x)

```

All the standard list processing functions can be obtained by partially parameterising foldr. Examples

```

sum = foldr (+) 0
product = foldr (*) 1
reverse = foldr postfix []
          where postfix a x = x ++ [a]

```

ZF expressions

ZF expressions give a concise syntax for a rather general class of iterations over lists. The notation is adapted from Zermelo Frankel set theory (whence the name “ZF”). A simple example of a ZF expression is:

```
[ n*n | n <- [1..100] ]
```

This is a list containing (in order) the squares of all the numbers from 1 to 100. The above expression would be read aloud as “list of all n*n such that n drawn from the list 1 to 100”. Note that n is a local variable of the above expression. The variable-binding construct to the right of the bar is called a “generator” — the “<-” sign denotes that the variable introduced on its left ranges over all the elements of the list on its right. The general form of a ZF expression in Miranda is:

```
[ body | qualifiers ]
```

where each qualifier is either a generator, of the form var<-exp, or else a filter, which is a boolean expression used to restrict the ranges of the variables introduced by the generators. When two or more qualifiers are present they are separated by semicolons. An example of a ZF expression with two generators is given by the following definition of a function for returning a list of all the permutations of a given list,

```

perms [] = [[]]
perms x = [ a:y | a <- x; y <- perms (x--[a]) ]

```

The use of a filter is shown by the following definition of a function which takes a number and returns a list of all its factors,

```
factors n = [ i | i <- [1..n div 2]; n mod i = 0 ]
```

ZF notation often allows remarkable conciseness of expression. We give two examples. Here is a Miranda statement of Hoare's "Quicksort" algorithm, as a method of sorting a list,

```
sort [] = []
sort (a:x) = sort [ b | b <- x; b <= a ]
           ++ [a] ++
           sort [ b | b <- x; b > a ]
```

Here is a Miranda solution to the eight queens problem. We have to place eight queens on chess board so that no queen gives check to any other. Since any solution must have exactly one queen in each column, a suitable representation for a board is a list of integers giving the row number of the queen in each successive column. In the following script the function "queens n" returns all safe ways to place queens on the first n columns. A list of all solutions to the eight queens problem is therefore obtained by printing the value of (queens 8)

```
queens 0 = [[]]
queens (n+1) = [ q:b | b <- queens n; q <- [0..7]; safe q b ]
safe q b = and [ ~checks q b i | i <- [0..#b-1] ]
checks q b i = q=b!i / abs(q - b!i)=i+1
```

Lazy evaluation and infinite lists

Miranda's evaluation mechanism is "lazy", in the sense that no subexpression is evaluated until its value is known to be required. One consequence of this is that it is possible to define functions which are non-strict (meaning that they are capable of returning an answer even if one of their arguments is undefined). For example we can define a conditional function as follows,

```
if True x y = x
if False x y = y
```

and then use it in such situations as "if (x=0) 0 (1/x)".

The other main consequence of lazy evaluation is that it makes it possible to write down definitions of infinite data structures. Here are some examples of Miranda definitions of infinite lists (note that there is a modified form of the "..." notation for endless arithmetic progressions)

```
ones = 1 : ones
repeat a = x
  where x = a : x
nats = [0..]
odds = [1,3..]
squares = [ n*n | n <- [0..] ]
perfects = [ n | n <- [1..]; sum(factors n) = n ]
primes = sieve [ 2.. ]
  where
    sieve (p:x) = p : sieve [ n | n <- x; n mod p > 0 ]
```

One interesting application of infinite lists is to act as lookup tables for caching the values of a function. For example our earlier naive definition of "fib" can be improved from exponential to linear complexity by changing the recursion to use a lookup table, thus

```
fib 0 = 1
fib 1 = 1
fib (n+2) = flist!(n+1) + flist!n
  where
    flist = map fib [ 0.. ]
```


Another important use of infinite lists is that they enable us to write functional programs representing networks of communicating processes. Consider for example the hamming numbers problem — we have to print in ascending order all numbers of the form $2^a 3^b 5^c$, for $a, b, c \geq 0$. There is a nice solution to this problem in terms of communicating processes, which can be expressed in Miranda as follows

```
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
  where
    f a = [ n*a | n <- hamming ]
    merge (a:x) (b:y) = a : merge x (b:y), a < b
                  = b : merge (a:x) y, a > b
                  = a : merge x y, a = b
```

Polymorphic strong typing

Miranda is strongly typed. That is, every expression and every subexpression has a type, which can be deduced at compile time, and any inconsistency in the type structure of a script results in a compile time error message. We here briefly summarise Miranda's notation for its types.

There are three primitive types, called `num`, `bool`, and `char`. The type `num` comprises integer and floating point numbers (the distinction between integers and floating point numbers is handled at run time — this is not regarded as being a type distinction). There are two values of type `bool`, called `True` and `False`. The type `char` comprises the ascii character set — character constants are written in single quotes, using C escape conventions, e.g.

If `T` is type, then `[T]` is the type of lists whose elements are of type `T`. For example `[[1,2],[2,3],[4,5]]` is of type `[[num]]`, that is it is a list of lists of numbers. String constants are of type `char`, in fact a string such as `"hello"` is simply a shorthand way of writing `['h','e','l','l','o']`.

If `T1` to `Tn` are types, then `(T1,...,Tn)` is the type of tuples with objects of these types as components. For example `(True,"hello",36)` is of type `(bool,[char],num)`.

If `T1` and `T2` are types, then `T1->T2` is the type of a function with arguments in `T1` and results in `T2`. For example the function `sum` is of type `[num]->num`. The function `quadsolve`, given earlier, is of type `num->num->num->[num]`. Note that `"->"` is right associative.

Miranda scripts can include type declarations. These are written using `"::"` to mean is of type. Example

```
sq :: num -> num
sq n = n * n
```

The type declaration is not necessary, however. The compiler is always able to deduce the type of an identifier from its defining equation. Miranda scripts often contain type declarations as these are useful for documentation (and they provide an extra check, since the typechecker will complain if the declared type is inconsistent with the inferred one).

Types can be polymorphic, in the sense of Milner [Milner 78]. This is indicated by using the symbols `***` etc as an alphabet of generic type variables. Example, the identity function, defined in the Miranda library as

```
id x = x
```

has the following type

```
id :: * -> *
```

this means that the identity function has many types. Namely all those which can be obtained by substituting an arbitrary type for the generic type variable, eg `"num->num"`, `"bool->bool"`,

“(→*) → (→*)” and so on.

We illustrate the Miranda type system by giving types for some of the functions so far defined in this article

```
fac :: num → num
ack :: num → num → num
sum :: [num] → num
month :: [char] → bool
reverse :: [*] → [*]
fst :: (*,**) → *
snd :: (*,**) → **
foldr :: (→**→**) → ** → [*] → **
perms :: [*] → [[*]]
```

User defined types

The user may introduce new types. This is done by an equation in “::”. For example a type of labelled binary trees (with numeric labels) would be introduced as follows,

```
tree ::= Nilt | Node num tree tree
```

This introduces three new identifiers — “tree” which is the name of the type, and “Nilt” and “Node” which are the constructors for trees. Nilt is an atomic constructor, while Node takes three arguments, of the types shown. Here is an example of a tree built using these constructors

```
t1 = Node 7 (Node 3 Nilt Nilt) (Node 4 Nilt Nilt)
```

To analyse an object of user defined type, we use pattern matching. For example here is a definition of a function for taking the mirror image of a tree

```
mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x)
```

User defined types can be polymorphic — this is shown by introducing one or more generic type variables as parameters of the “::” equation. For example we can generalise the definition of tree to allow arbitrary labels, thus

```
tree * ::= Nilt | Node * (tree *) (tree *)
```

this introduces a family of tree types, including tree num, tree bool, tree(char→char) etc.

The types introduced by “::” definitions are called “algebraic types”. Algebraic types are a very general idea. They include scalar enumeration types, eg

```
color ::= Red | Orange | Yellow | Green | Blue | Indigo | Violet
```

and also give us a way to do union types, for example

```
bool_or_num ::= Left bool | Right num
```

It is interesting to note that all the basic data types of Miranda could be defined from first principles, using “::” equations. For example here are type definitions for bool, (natural) numbers and lists,

```
bool ::= True | False
nat ::= Zero | Suc nat
list * ::= Nil | Cons * (list *)
```

Having types such as “num” built in is done for reasons of efficiency — it isn’t logically necessary.

It is also possible to associate “laws” with the constructors of an algebraic type, which are applied whenever an object of the type is built. For example we can associate laws with the Node constructor of the tree type above, so that trees are always balanced. We omit discussion of this feature of Miranda here for lack of space — interested readers will find more details in the references [Thompson 86, Turner 85].

Type synonyms

The Miranda programmer can introduce a new name for an already existing type. We use “==” for these definitions, to distinguish them from ordinary value definitions. Examples

```
string == [char]
matrix == [[num]]
```

Type synonyms are entirely transparent to the typechecker — it is best to think of them as macros. It is also possible to introduce synonyms for families of types. This is done by using generic type symbols as formal parameters, as in

```
array * == [[*]]
```

so now eg ‘array num’ is the same type as ‘matrix’.

Abstract data types

In addition to concrete types, introduced by “::=” or “==” equations, Miranda permits the definition of abstract types, whose implementation is “hidden” from the rest of the program. To show how this works we give the standard example of defining stack as an abstract data type (here based on lists):

```
abstype stack *
with empty :: stack *
    isempty :: stack * -> bool
    push :: * -> stack * -> stack *
    pop :: stack * -> stack *
    top :: stack * -> *

stack * == [*]
empty = []
isempty x = (x==[])
push a x = (a:x)
pop (a:x) = x
top (a:x) = a
```

We see that the definition of an abstract data type consists of two parts. First a declaration of the form “abstype ... with ...”, where the names following the “with” are called the *signature* of the abstract data type. These names are the interface between the abstract data type and the rest of the program. Then a set of equations giving bindings for the names introduced in the abstype declaration. These are called the *implementation equations*.

The type abstraction is enforced by the typechecker. The mechanism works as follows. When typechecking the implementation equations the abstract type and its representation are treated as being the same type. In the whole of the rest of the script the abstract type and its representation are treated as two separate and completely unrelated types. This is somewhat different from the usual mechanism for implementing abstract data types, but has a number of advantages. It is discussed at somewhat greater length in [Turner 85].

Separate compilation and linking

The basic mechanisms for separate compilation and linking are extremely simple. Any Miranda script can contain one or more directives of the form

```
%include "pathname"
```

where *pathname* is the name of another Miranda script file (which might itself contain include directives, and so on recursively — cycles in the include structure are not permitted however). The visibility of names to an including script is controlled by a directive in the included script, of the form

```
%export names
```

it is permitted to export types as well as values. It is not permitted to export a value to a place where its type is unknown, so if you export an object of a locally defined type, the typename must be exported also. Exporting the name of a “`::=`” type automatically exports all its constructors. If a script does not contain an export directive, then the default is that all the names (and typenames) it defines will be exported (but not those which it acquired by a `%include` statement).

It is also permitted to write a *parameterised script*, in which certain names and/or typenames are declared as “free”. An example is that we might wish to write a package for doing matrix algebra without knowing what the type of the matrix elements are going to be. A header for such a package could look like this:

```
%free { element :: type
      zero, unit :: element
      mult, add, subtract, divide :: element->element->element
}

%export matmult determinant eigenvalues eigenvectors ...
-- here would follow definitions of matmult, determinant,
-- eigenvalues, etc. in terms of the free identifiers zero,
-- unit, mult, add, subtract, divide
```

In the using script, the corresponding include statement must give a set of bindings for the free variables of the included script. For example here is an instantiation of the matrix package sketched above, with real numbers as the chosen element type:

```
%include "matrix_pack"
{ element == num; zero = 0; unit = 1
  mult = *; add = +; subtract = -; divide = /
}
```

The three directives `%include` `%export` `%free` provide the Miranda programmer with a flexible and type secure mechanism for structuring larger pieces of software from libraries of smaller components. (Note: the `%free` directive is not yet implemented in the version of Miranda currently being distributed)

Separate compilation is administered without user intervention. Each file containing a Miranda script is shadowed by an object code file created by the system, and object code files are automatically recreated and relinked if they become out of date with respect to any relevant source. (This behaviour is strongly analogous to that achieved by the UNIX program `make`, except that here the user is not required to write a makefile — the necessary dependency information is inferred from the `%include` directives in the Miranda source.)

Current implementation status

The Miranda system is available for (at least) the following machines under Berkeley UNIX:- VAX, SUN 2, SUN 3, ORION, Gould, Apollo. This is an interpretive implementation which works by compiling Miranda scripts to an intermediate code based on combinators. It is currently running at 130 sites. Licensing information can be obtained from the net address "mira-request%ukc.ac.uk@ucl-cs.arpa" or (usenet) "mcvax!ukc!mira-request" or by real mail from

Research Software Limited
23 St Augustines Road
Canterbury
Kent CT1 1XP
England
telephone: +44 227 471844

At the time of writing a port to system V is under study, and may well be accomplished by the time this paper appears. Also under study (for appearance on a somewhat longer timescale) is the possibility of native code compilers for Miranda for a range of machines, to provide a much faster implementation.

REFERENCES

- Milner, R. "A Theory of Type Polymorphism in Programming" *Journal of Computer and System Sciences*, vol 17, 1978
- Thompson, S.J. "Laws in Miranda" *Proceedings 4th ACM International Conference on LISP and Functional Programming*, Boston Mass, August 1986
- Turner, D.A. "Miranda: A non-strict functional language with polymorphic types" *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy France, September 1985 (Springer Lecture Notes in Computer Science, vol 201).

[Note: an earlier draft of this paper appeared in SIGPLAN Notices, December 1986]

Experiences with DREGS

Allan Bricker

bricker@bleu.wisc.edu

Morgan Clark

morganc@poona.wisc.edu

Tad Lebeck

tadl@brie.wisc.edu

Barton P. Miller

bart@asiago.wisc.edu

Peter Wu

pwu@trochos.wisc.edu

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

DREGS is a system for building multi-player distributed games. A DREGS game consists of one copy of the game at each player's site and a central arbiter through which all communication passes. Games are described by a set of objects and the events to update those objects. Building a new game involves defining objects, events, and writing procedures to handle the events. DREGS makes creating distributed games almost as easy as creating single player games.

Recently we used DREGS to write three new games, Mazewar, Dhack, and Xtrek. In Mazewar players wander about in a maze shooting at other players. Dhack is a real-time version of Hack where players explore a dungeon, interacting with one another, fighting monsters along the way, and collecting treasure. Xtrek is a real-time game based on the games Empire, Trek, and Conquest where players maneuver starships in a galaxy filled with friendly and hostile planets and other players.

These new games are more complex than games previously implemented with DREGS. We encountered the need for shared dynamic data structures and autonomous players that run independently from other players. The monsters in Dhack are an example of autonomous players. We were able to overcome these new problems within facilities provided by DREGS.

1. Introduction

Distributed Runtime Environment for Game Support, DREGS[1], is a system for supporting multi-player distributed games. It currently supports games such as Pline (a 10-way talk program), Tank, and SpaceWar. We have recently implemented two new games, Mazewar and Dhack, using DREGS and are converting an existing game, Xtrek. The new games are more complex than previous DREGS games and have features that stretch the abilities of DREGS.

We first describe the features and structures of DREGS and then describe the new games. Next we present our experiences with DREGS and discuss extensions and additions that we made to implement the new games.

2. DREGS Overview

Dregs is a system used for writing multi-player, distributed games. We can examine DREGS from several perspectives: game model, game organization and structure, building a DREGS game, and DREGS implementation.

2.1. Game Model

The DREGS model of a distributed game consists of several identical copies of the game and one central *arbiter* to coordinate the copies. Each copy of the game is divided into an *Input Manager*, a *Game Manager*, and an *Output Manager* (see Figure 1). Each game site communicates exclusively with the arbiter; no two game sites communicate directly.

Each game site has a collection of *objects* which represent the state of the game. There are two types of objects: those created by a local game site, called *native objects*, and those created at remote game sites, called *clone objects*. Every game manager has a copy of every object in the game. Both types of objects accept *events* that cause the objects to be updated.

The input manager is responsible for accepting input from a player and packaging that input into events that will eventually be delivered to all game managers. The game manager receives the events generated by the input managers and modifies its objects based on those events. The game manager then checks all its native objects for interactions with other objects, possibly producing additional events. Any new events are then sent to the arbiter. Finally the game manager sends all its display output to the output manager. DREGS does not actually provide an output manager, but routines from other display packages such as X[2] and curses[3] can be used for display.

The arbiter is responsible for ensuring that every game site has a consistent view of the game and that all game sites execute synchronously. The arbiter collects events from all input and game managers for a fixed time quantum, then broadcasts them to every game manager. This keeps all game sites consistent and synchronized with each other. When a new game site wishes to join a running game, the arbiter

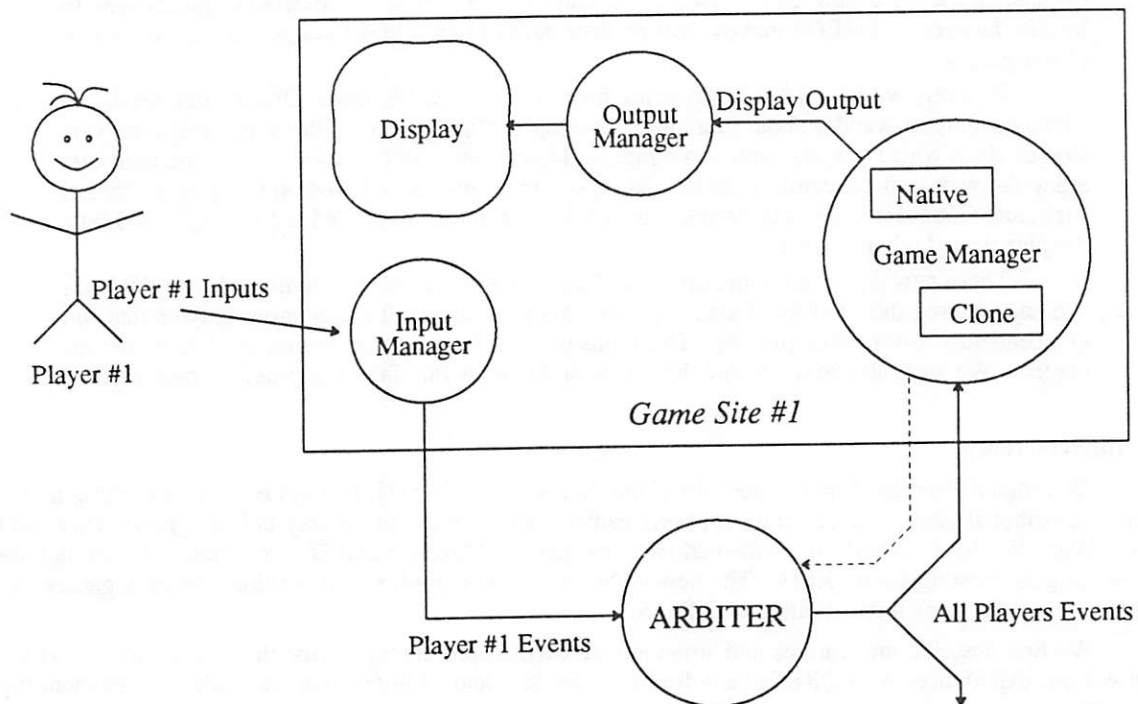


Figure 1: DREGS Communication Paths

stops transmitting messages while it sends a dump of the current objects to the new player. Once the new player has a consistent copy of the game objects, the arbiter resumes transmitting events, sending to the new game site as well as the old game sites.

2.2. Game Organization and Structure

The two important attributes of any DREGS game are its object and event definitions. An entire object is usually defined in a single definition, containing all the relevant information for that object. Objects are usually distributed in their entirety to every game site. An event definition is in two parts. There is a unique identifier for each event and an action function associated with each event identifier. For example, in Dhack an event specifying a player is moving to the left consists of a definition of the move-left event identifier and the definitions of functions to decrease that character's X coordinate by one and to check if that character is going to hit something like a trap or a wall.

DREGS game sites communicate entirely through events. An event is composed of several parts. First comes a short header specifying the player originating the event, the object type and the specific object associated with this event, and the event type. In some cases, particularly for most input events, that is all the information contained in the event. In other cases, there will follow an arbitrary amount of data relevant to the event.

The DREGS model describes four different event types. *Input events* result from input actions like key strokes. *Replicated procedure events* are generated by the game managers in response to other events. *Generic object events* are defined for every object type. These events include events to create, destroy, update, and display objects. Last, *interval timer events* occur at regular intervals as the result of a timer. Input events, replicated procedure events, and generic object events are distributed to every game site; interval timer events are generated and executed locally at a particular game site. Interval timer events that cause any changes to the state of the game result in replicated procedure events being generated and distributed.

The DREGS model divides a game into three parts: an input manager, a game manager, and an output manager. A game implementation is divided differently: into a main program, an input manager, and a game manager with an interface to an output manager. A person wishing to play a DREGS game runs the main program. It then forks (creates processes) the input and game managers. The input manager and game managers run as independent processes, communicating only with the arbiter.

The input manager waits for input from the user's keyboard or other device (such as a mouse) and generates the appropriate event for each accepted input. The input manager is the only part of a DREGS game that is not explicitly synchronized with the arbiter. The input manager sends events to the arbiter as they are received.

The game manager and output manager interface makes up the other part of a DREGS game. The game manager receives events from the arbiter and performs the appropriate actions. The output manager is called by the game manager through procedures in standard libraries.

2.3. Building a DREGS Game

Building a new game requires defining objects, events, procedures to handle events, and output routines. The first step in writing a DREGS game is to define the objects. Objects are defined by specifying their attributes. The next step is to define input events and replicated procedure events. Defining an input event means inserting code in the input manager to send an input event to the arbiter for a given player input. Defining a replicated procedure event means writing a procedure in the game manager to assemble the data structure for the event and send it to the arbiter. After the events are defined, routines must be written in the game manager to handle these events. For example, in Mazewar the input manager generates a "fire" event when the player hits the space bar. When a game manager receives the fire event, it checks to see if the player is hit, and if so it generates a replicated procedure event telling all game managers. The last step is to write the output routines to display the proper view for the player. This usually involves calling library routines to plot graphics or to display characters.

At present, the easiest way to create a new game is to modify an existing game. We used Tank[1] to build Mazewar and Dhack. Eventually new DREGS games will be designed by writing the game in a Game Description Language (GDL). Using GDL, the designer would specify the object and event

definitions for the game along with code to be executed for each event. The GDL description would then be compiled to produce source code for the game in C, which can then be compiled to produce the executable game. The GDL compiler would automatically generate the routines to create input and replicated procedure events and routines to communicate with the arbiter. A GDL compiler is currently being built.

2.4. DREGS Implementation

DREGS is written in C. The current implementation runs on 4.3BSD UNIX on Microvax-II and IBM RT/PC workstations and can support up to ten players per game. DREGS games exchange information using XDR, Sun Microsystem's External Data Representation Protocol[4], to allow DREGS games to work across heterogeneous environments. Each event is encoded as a separate XDR record. All our games currently use X routines for graphics output.

3. Game Descriptions

We have implemented three new games in DREGS. Mazewar is an implementation of the familiar game Mazewars, Dhack is a distributed version of the adventure game Hack, and Xtrek is a port of the UNIX game of the same name.

3.1. Mazewar

Mazewar is a familiar game that has had several previous implementations. In Mazewar players maneuver eyeballs through a maze trying to shoot at other players and to avoid being hit by other players. A player can only see, move, and fire in the direction his or her eyeball is facing. Eyeballs can move forward, turn left or right, peek left or right around a corner in the maze, or fire. When a player hits another player's eyeball he or she receives one score point. When an eyeball is hit, it sustains a unit of damage and is teleported to a different place in the maze. Each eyeball can sustain ten units of damage before it dies. The display for Mazewar is in three parts: a three-dimensional graphic display of the corridor the player can see, a plan display of the entire maze showing the player's location, and some status information (see Figure 2).

3.2. Dhack

Dhack is a distributed, real-time version of the adventure game Hack. Dhack has several features similar to regular Hack. Players explore a dungeon of many levels, each level getting more difficult than the previous levels. There are monsters that roam the maze that are generally hostile to players. As players kill monsters they gain experience and get stronger. Players can collect various forms of treasure and gold scattered around the dungeon floor. Players can also engrave messages in the dungeon floor. There are traps and secret doors and hallways for which players must search. The display for Dhack (see Figure 3) strongly resembles that of regular Hack. Another region will be added to the bottom of the display for status information and messages.

Dhack differs from Hack in several important ways. The most important difference is that all players in the game operate in the same dungeon. Players can interact with each other in several ways. They can see each other, send messages to each other, cooperate with each other to fight monsters, or even fight each other.

Another difference from Hack is that Dhack moves in real time. Players can move at whatever speed they desire, independent of each other. The only enforced restriction is that only one move per arbiter interval is permitted. Monsters move independently of any player. If a player stops to rest, a monster can walk up and beat that player to death unless the player takes action. Monster movement speed depends on the monster type. Typically stronger monsters will move faster than weaker monsters.

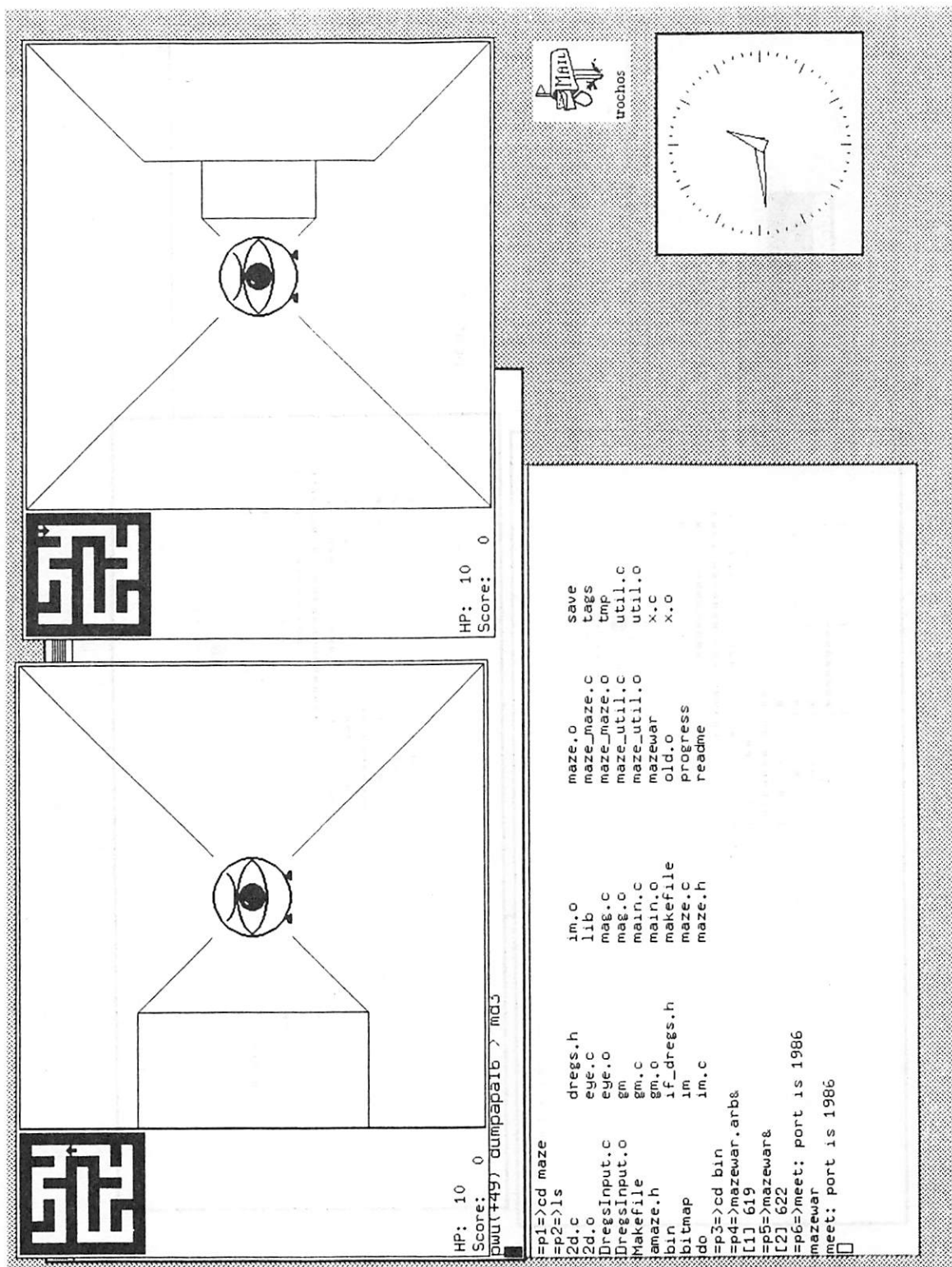


Figure 2: Mazewar shown with two players on the same screen

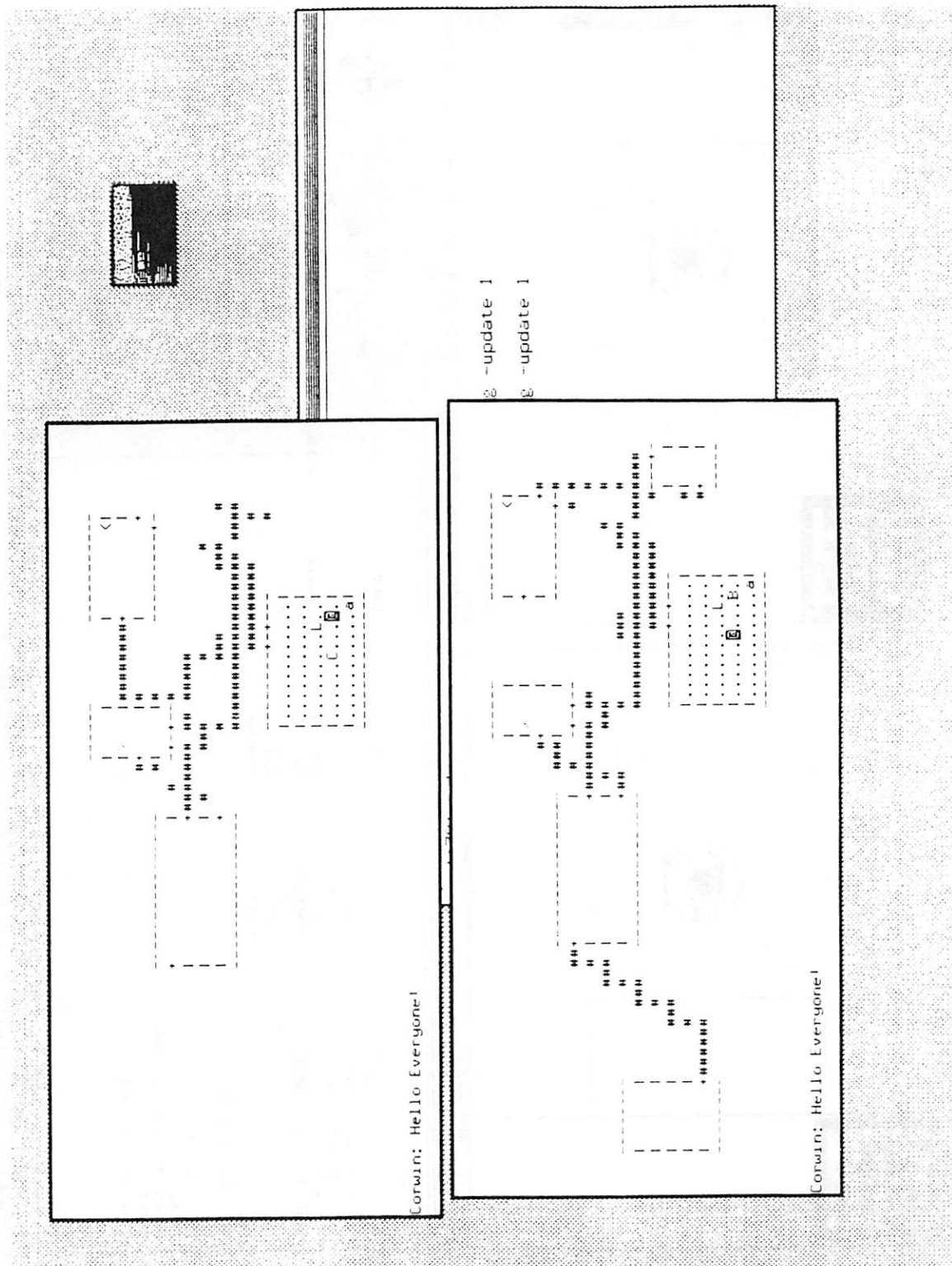


Figure 3: Dhack shown with two players on the same screen

A third difference is that a particular game will continue to exist even if players come and go. A dungeon level is created only once; the game first checks to see if the level has already been created any time a player wants a new level. If it has, the old level will be used instead of a new one being created. The objects and monsters that had been on that level are restored. After every player has left a particular dungeon level, that level is saved with all its objects and monsters. The monsters on that level then become dormant, waiting for an active player to return to that level.

When a player wants to save the game, instead of removing that player from the game altogether, a player statue is created in place of the player. This statue is then treated just like any other object, i.e. it can be picked up, carried, and dropped. However, it cannot be destroyed or damaged. Then when that player wants to rejoin the game, the player character is restored at the current position of the statue. This could be different position from where the player was when he or she saved the game.

3.3. Xtrek

Xtrek[5] is a real-time multi-player game based on the game Empire, with many ideas also taken from the Unix game Trek83 and the VMS game Conquest. Xtrek uses shared memory segments to allow multiple players to update a common data structure. We have found Xtrek to be a popular and entertaining game, however it causes too much load on the host machine on which it is played. We are porting Xtrek to DREGS to alleviate the load that results from several game processes running on a single processor. We find converting Xtrek to run under DREGS to be a straight-forward task. We believe that Xtrek is especially suited for DREGS because it is highly interactive and event-action oriented.

4. Experiences with DREGS

We found DREGS to be a useful and simple tool to implement multi-player, distributed games. Designing a distributed game is only slightly more difficult than designing a single-host game. In this section we comment on useful features of DREGS, the difficulties we encountered using DREGS, and the solutions we found for those difficulties.

4.1. Useful Features

The most helpful features of DREGS for game designers are the communication and synchronization facilities provided by the arbiter and the DREGS structure. We designed Dhack and Mazewar from an existing DREGS game, Tank. Like all DREGS games, Tank runs in three parts: the main program and the input and game managers. The main program does all the work to connect to the arbiter, passing the resulting socket descriptors to the input and game managers. The input and game managers in Tank had functions we could use to send data to and from the arbiter, using the XDR record streams. All that remained was to write the routines to call the correct XDR functions to send our data to and from the arbiter. The synchronization features of DREGS also eased the game design task. DREGS guarantees that every game site will see the same events in the same order. We did not have to worry about keeping each game site consistent, so we could write our games while almost ignoring their distributed aspect.

4.2. New Problems

The original DREGS model lacked several features we needed to build our new games. First, both Dhack and Mazewar needed shared dynamic data structures that did not belong to any regular player. Second, there was no way to cause events for objects not related to any players. Third, starting a DREGS game was difficult. The person starting a DREGS game had to specify the location of a running arbiter to which the game could connect. If there was no running arbiter, then the player had to manually start one.

4.2.1. Shared Data Structures

All our games have playing fields common to every player. In previous games, such as Tank, the playing field was fixed at compile time and never changed. The maze in Mazewar is created at startup time, but it does not change thereafter. The dungeon in Dhack changes dynamically during a game. In both new games, the maze is a DREGS object. This introduces problems with creation and ownership of the maze objects. The maze should not belong to any single player, since it is shared by every player in the game. Therefore it should not be an object created and owned by any ordinary player. Furthermore, when a player quits a game, all of that player's objects are destroyed, so the maze should not belong to any

player that can quit the game. We solved these problems in two ways. In Mazewar, every player owns a copy of the maze, and in Dhack, every maze level is owned by a special player, called an *autonomous player*.

In Mazewar, the maze does not change over time, so we generate the maze from a random seed. The first player in a game creates a seed and stores it as a native object. Each subsequent player receives the existing seed and creates a copy. Each game site uses the seed to generate its copy of the maze. The amount of information that must be passed around to generate a maze is small. This works well for mazes that do not change over time.

In Dhack, the structure of the maze can change over time so the scheme used in Mazewar cannot be used. Our solution was to create an autonomous player with complete responsibility for creating and distributing the dungeon levels. The autonomous player appears to the rest of the game as any other player, but it behaves in a manner different from other players. Since every event is distributed to every game site, the autonomous player is written to respond to some events to which ordinary players do not. For example, when a player wants to enter a new level, that player creates and sends a replicated procedure event asking for a new level. Regular players do not respond to this particular event, but the autonomous player will create the desired level and distribute it. The autonomous player also differs from regular players in that it does not accept player input and does not generate any output. Its game site consists of only a game manager; there is no associated input manager or output manager (see Figure 4). For technical reasons we had to create an input manager, but it simply connects to the arbiter and goes to sleep.

4.2.2. Autonomous Objects: Monsters

A second problem resulted from the monsters in Dhack. Like dungeon levels, monster also do not logically belong to any player, but unlike the dungeon levels they are mobile and should be able to move on their own. Some player needs to generate movement events for monsters. Some player other than the regular players should have responsibility for the monsters.

An obvious candidate was the autonomous player. When the autonomous player generates a new level, it creates new monsters and distributes them. It also creates new monsters as old ones are destroyed. Thereafter, the autonomous player generates movement events for each currently-moving monster. The monster events are independent of any other players' actions.

4.2.3. Arbiter Startup and Connection

Before any player can start playing a game, the arbiter and autonomous player, if appropriate, must be running in a known place. A player starting a new game cannot simply assume the arbiter is running. The machine on which it had been running may have crashed, or the arbiter may have been manually stopped. The player starting to play may not even know where the arbiter is running. The new player should be able to look for an arbiter and start one running if necessary. Once an arbiter is running it should continue running until there are no players actively playing the game. If the player that started up the arbiter subsequently quits the game, the arbiter should continue to run. The problems are, first, to find and connect to an arbiter if one is running, and, second, to start an independent arbiter if none is running.

We believe that the ideal solution to the problem of connecting to the arbiter is to have the game and the arbiter meet through a connection server[6]. When the arbiter is started, it posts its address to the connection server. A new player first asks the connection server for the address of an arbiter. The connection server supplies the address of the arbiter and the game can proceed. This eliminates the need to know the location of the arbiter or the players.

This scheme can insure that only one arbiter is started. Our connection server will accept only one connection from a given server. Later requests to post an address for a server of the same name are rejected. When each player starts running, it tries to connect to the connection server calling itself the arbiter. Failure means that there is already an arbiter running, and the player can use the running arbiter. Success means that there is no arbiter running and one should be started.

4.2.4. Objectless Events

A requirement that every event must correspond to a particular object caused a problem when a Dhack game was started. The autonomous player performs all character creation, and each game must ask

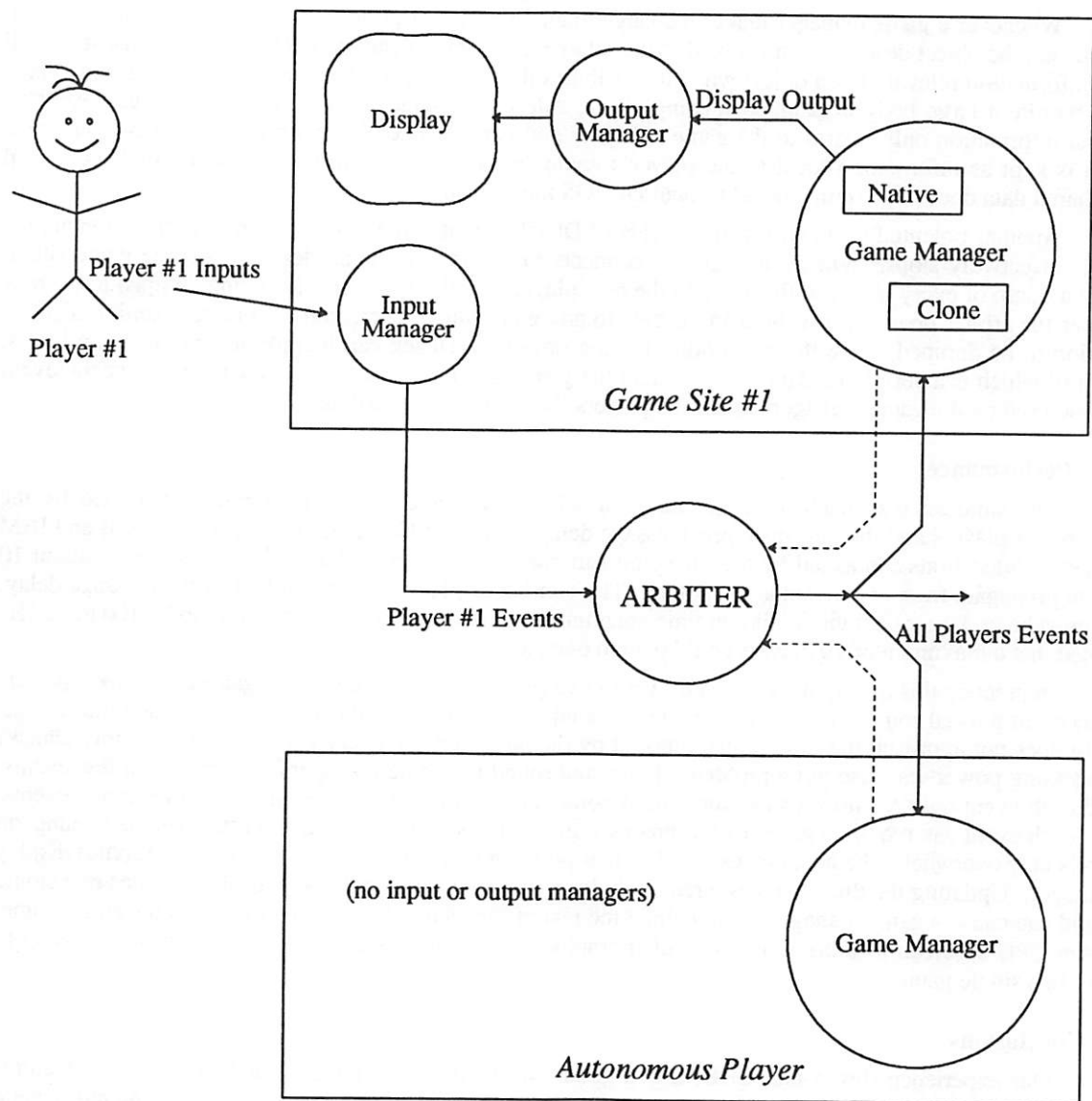


Figure 4: Dhack Communication Paths

the autonomous player to restore its character from a disk file for a saved player or create it for a new player. However, the only messages DREGS will send are events, and the character associated with the create character request does not yet exist.

The solution was to create a dummy character object. Immediately after a player starts, it creates a dummy character object and asks the autonomous player to create or restore the real character. The autonomous player then fills in the dummy with the character data and distributes the new character to every

player.

4.2.5. Other Difficulties

Whenever a game manager makes a change in an object description that no other game manager can duplicate, the object description must be distributed to every game manager. DREGS usually distributes all the information relevant to an object when it distributes the object itself. Each player's character in Dhack can require a large body of data, so sending all the data for a character can lead to very large messages. Some information only relates to the game's display and is not needed by any other game manager. This data is kept as information local to the player's game site and is not distributed to other players. Local unshared data does not require any additional DREGS mechanisms.

Another potential problem for the players of Dhack is that while a new player is joining a game the game effectively stops. When a new player connects to the arbiter, the arbiter requests one game site to send a dump of every object in the game to the new player. While the objects are being dumped to the new player the arbiter does not distribute any events to any game sites. There can be a large quantity of information to be dumped, since the collection of game objects in Dhack can include several dungeon levels, each of which is a very large data structure, and the game may be stopped for a long time. A special event can be used by the game managers to tell the players that there may be a delay.

4.3. Performance

The smallest reasonable time quantum for which the arbiter generates a round is limited by the number of players and the inter-machine message delay. Our DREGS games run on Microvax-II and IBM RT/PC workstations connected by a 10 megabit Ethernet. We observed that each message takes about 10 ms to propagate from one machine to another. The number of players, N , multiplied by the message delay, D , must be no larger than the minimum time quantum for a round. Setting the quantum to be 100 ms determined that a maximum of 10 players could play in one game.

In practice this quantum works well. Up to five players can play any of the games with good performance. In a usual round most game sites do not send any messages to the arbiter, so message traffic typically does not approach the time limits imposed by the hardware. In every game (except possibly Dhack) processing power was also not a problem. In a usual round the game managers only receive a few events, and each event only requires a small amount of computation. In Dhack, a round can involve many events, and each event can require a great deal of processing, so games with many players active on many dungeon levels may overwhelm the host processor. The real problem in all of our games is with the graphics display manager. Updating the display takes a relatively long period of time, and many display updates in a single round can cause a game manager to lag behind the rest of the game, thereby slowing down the entire game. Faster CPU's, screen memories, and dedicated graphics display processors will allow ten or more player to play in a single game.

5. Conclusions

Our experience shows that DREGS is a good tool for developing distributed games. New features can be easily added to a game structure by creating new objects and events. We encountered several unsolved problems, but each one could be solved without fundamental changes to DREGS. New DREGS features such as the autonomous player can be added with a minimum amount of work. An existing DREGS game serves as a good base for designing a new game. The object and event descriptions need to be changed, but the DREGS game structure is already in place. Since DREGS handles communication between multiple players, the remaining difficulty in designing new games is designing the graphic display and user interface.

6. References

- [1] Allan Bricker, Tad Lebeck, and Barton P. Miller, "DREGS: A Distributed Runtime Environment for Game Support," *Proc. of the 1986 Eur. Unix Users Group Conf.*, Manchester, England, (September 1986).
- [2] J. Gettys, R. Newman, and T. D. Fera, "Xlib - C Language X Interface," X Reference Manual (January 1986).

- [3] K. Arnold, "Screen Updating and Cursor Movement Optimization, A Library Package," in *UNIX Programmer's Manual*, 0.
- [4] B. Lyon, "Sun External Data Representation Protocol Specification," Sun Microsystems, Inc. Technical Report (April 1985).
- [5] Chris Guthrie, "Xtrek - A Multi-player, Space Shoot-'em-up Game," Xtrek User Documentation (1986).
- [6] D. Draheim, B.P. Miller, and S. Snyder, "A Reliable and Secure UNIX Connection Service," *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, (March 1987).

